SPECTRUM: Iniciación al CODIGO MAQUINA

Antonio BELLIDO

PARANINFO S.A.

900

SPECTRUM: Iniciación al

CODIGO MAQUINA

Antonio BELLIDO

SPECTRUM: Iniciación al CODIGO MAQUINA

1985

PARANINFO S.A.

MADRID

© ANTONIO BELLIDO
Madrid (España)

Reservados los derechos para todos los países. Ninguna parte de esta publicación, incluido el diseño de la cubierta, puede ser reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste electrónico, químico, mecánico, electro-óptico, grabación, fotocopia o cualquier otro, sin la previa autorización escrita por parte de la Editorial.

IMPRESO EN ESPAÑA
PRINTED IN SPAIN

Depósito legal: M. 35.886-1985
ISBN: 84-283-1421-7

PARANINFO S.A. Magallanes, 25 - 28015 - MADRID (3-3437)

Impreso en Lavel. Los Llanos, nave 6. Humanes (Madrid)

Indice de materias

Introducción 7
Código máquina 8
Lenguaje de asamblea 10
Ensamblador 11
Microprocesador 12
Memoria 15
Puesta en marcha 18
Proceso general de trabajo 21
Cargando los registros 23
Usando los registros 28
Sumando y restando 30
Saltando 34
Subrutinas 42
Pseudo-operaciones (Primera parte) 47
El primer gráfico 49
La primera letra 51
Un poco de acción 54
A través del teclado 58
Pseudo-operaciones (Segunda parte) 67
La pila 76
Bucles 79
Las señalizaciones. El registro f 80
Un poco de matemáticas. 8 bits 82
Un poco más de matemáticas. 16 bits 89
Manipulando bits 91
Penúltimo 97
Todo comienza aquí 99
Conclusión 101
Apéndices 102
- Subrutinas en la ROM 102
- Atributos 103
- Códigos de control 104
- Juego de instrucciones del Z80 105
- Interpretaciones abreviadas de memoria 110
- Tabla de conversión decimal y hexadecimal 111

Introducción

Non Plus Ultra

En términos de programación nada *hay más allá* del código máquina.

El código máquina implica *hablar cara a cara* con el cerebro del computador: EL MICROPROCESADOR (μP).

El microprocesador es la semilla de la revolución en la cual estamos inmersos, así que parece razonable aprender a comunicarnos directamente con él.

A través de un microprocesador se pueden realizar secuencias de acciones alternativas que, extremando la idea, podrían controlar un robot de características humanas. Sin ir tan lejos, en este libro vamos a comunicarnos con el μP gracias al teclado y a una pantalla de televisión.

Puesto que el microprocesador va a ser nuestro interlocutor, bueno será saber a grandes rasgos qué es y qué es lo que hace.

Un microprocesador es un circuito integrado –con un cierto parecido a una cucaracha por su color y sus patas– que controla los flujos de corriente que le llegan.

Con respecto a lo que hace, la respuesta es bien sencilla:

– Suma dos números.

– Compara dos números.

– Memoriza números.

– Controla cuál es la siguiente cosa que debe hacer.

La misión de un programador es poner los números correctos en la secuencia adecuada.

Si Vd. ha decidido entrar en el macrocosmos del microprocesador, debe creerme si le digo que, por la misma sencillez del microprocesador, la programación en lenguaje-máquina es una técnica sencilla. Este libro pretende demostrarlo.

Su voluntad de aprender y mi deseo de mostrarle el camino, deben permitirnos avanzar juntos hasta la última línea.

Para evitar eventuales desfallecimientos, dos recomendaciones:

– Si algo no lo entiende *ahora,* siga y ya lo entenderá más adelante.

– Su interlocutor (el μP) es lógico y paciente. La paciencia y la lógica, pues, son sus armas.

Adelante, ¿quién dijo miedo?

Código máquina

Un computador está regido por su microprocesador que es el verdadero "cerebro" del sistema y que en el caso del Spectrum es el Z80A.

Este microprocesador, como cualquier otro, sólo puede procesar *unidades elementales de información.*

Las unidades elementales de información son, simplemente, números binarios.

El sistema de numeración binario –o en base dos– tiene, como parece obvio, dos símbolos básicos (o dígitos): el **1** y el **0**, también llamados **bits.**

La palabra **bit** es una contracción de los vocablos ingleses *binary digit* que, en español, significa dígito binario.

El hecho físico que exige que un computador sólo pueda interpretar números binarios, está basado en dejar pasar –o no– impulsos eléctricos a través de diminutos interruptores. Cuando un impulso pasa, admitimos un 1, en caso contrario un 0.

Con un interruptor sólo podremos generar dos números binarios: el 1 y el 0. El primero cuando dejamos pasar corriente por el interruptor y el segundo cuando la cortamos. En otras palabras, con un solo interruptor los números binarios posibles coinciden con los dígitos binarios.

Si suponemos dos circuitos, cada uno de ellos similar al otro, los números binarios que en estas condiciones podríamos representar, serían:

– los dos interruptores apagados equivale a 00

– el primero apagado y el segundo encendido, equivale a 0 1

– el primero encendido y el segundo apagado, equivale a 1 0

– los dos interruptores encendidos equivale a 1 1

Siguiendo esta línea de razonamiento se podría pensar, por ejemplo, en seis interruptores, con lo cual podríamos obtener números binarios del porte de

0 0 0 0 0 0 ó 0 1 0 0 1 0 ó 1 1 1 1 1 1

según la combinación de interruptores encendidos y apagados que hubiéramos elegido.

En función de todo lo expuesto en los últimos párrafos, podremos deducir que un microprocesador de 8 bits –como es el caso del Z80– puede manejar números binarios de 8 dígitos, es decir, cualquier combinación de ocho *unos* y *ceros*, como por ejemplo: 0 0 1 1 1 0 0 0 y es precisamente a este concepto al que se denomina y conoce como **byte** (se pronuncia bait) u octeto.

Y, puesto que el Z80 sólo puede manejar números binarios compuestos por 8 bits, las unidades elementales de información tendrán que ser *bytes* para que puedan ser procesadas por este microprocesador.

Y una secuencia de *bytes* es, en definitiva, el código máquina.

La labor que tenemos por delante es poner el byte adecuado en el lugar correcto.

En este sentido, debemos distinguir entre unidades elementales de información destinadas a ordenar algo al microprocesador, de aquellas otras que son simplemente datos. A las primeras las denominamos microinstrucciones.

Por razones de orden práctico, las unidades elementales de información se suelen representar por el número decimal o hexadecimal equivalente al byte que nos ocupe, así, en la página 183 del Manual del Spectrum podemos ver, en las columnas encabezadas por *código* y *Hex*, los números decimal y hexadecimal correspondientes a los binarios representativos de las microinstrucciones que acepta el microprocesador Z80. Veamos un ejemplo.

La primera de todas las microinstrucciones es la 0, así, si en el contexto de un programa, nosotros quisiéramos ordenarle al microprocesador que no operase, escribiríamos 0 0 0 0 0 0 0 0. El Z80, al leer esta microinstrucción, pasaría por encima de ella sin hacer absolutamente nada.

Para concluir este capítulo, diremos que programar en código máquina es escribir microinstrucciones y datos en forma de bytes de forma tal que al ser procesadas se obtengan los resultados esperados.

Lenguaje de asamblea

En el capítulo anterior vimos cómo un programa en código máquina es una secuencia de bytes o, dicho de otra forma, una lista –a veces larguísima– de números binarios de 8 dígitos (bytes).

Es fácil intuir que es prácticamente inhumano programar directamente en código máquina y, por esta razón, lo normal es escribir los programas en **lenguaje de asamblea.**

Cada instrucción en lenguaje de asamblea tiene su microinstrucción equivalente en código máquina, pero ya no es un número sino un **mnemotécnico.**

Los mnemotécnicos son contracciones de vocablos ingleses que rememoran la función que cumplen las microinstrucciones del código máquina. Los mnemotécnicos son conocidos como **nemónicos.**

En la columna cuarta del Apéndice A –página 183 del Manual del Spectrum– figuran los nemónicos del lenguaje de asamblea utilizados por el Z80 y al final de este libro encontrará una tabla con el juego completo.

Como ya quedó dicho más atrás, un microprocesador sólo puede interpretar números binarios y esto implica que un programa escrito en lenguaje de asamblea necesite ser transcrito a código máquina para que sea operativo.

Esta transcripción la puede hacer el propio programador, apoyándose en las tablas citadas, las cuales le permitirán poner en forma numérica las diferentes instrucciones que haya utilizado en lenguaje de asamblea. Los datos deberán pasarse a la misma base de numeración que hayamos utilizado para las microinstrucciones.

Esto no es lo usual y, en todo caso, no es el objetivo de este libro.

Nosotros aprenderemos a programar en lenguaje de asamblea, a traducirlo a código máquina gracias a un programa **ensamblador** y a almacenarlo en la zona de memoria adecuada para poderlo utilizar como subrutina en código máquina de programas escritos en BASIC.

¿Y qué es un ensamblador?

Ensamblador

Un ensamblador es un programa cuya misión es transcribir –de forma automática– un programa escrito en lenguaje de asamblea a su equivalente en código máquina.

En el mercado existen varios ensambladores y cada uno somete al usuario a diferentes reglas para su correcto uso, las cuales figuran detalladamente en sus respectivos manuales.

Estas reglas no varían mucho de un ensamblador a otro, pero el programador en lenguaje de asamblea debe conocer a fondo las del ensamblador que haya decidido utilizar, con el fin de sacarla el máximo partido posible.

Nosotros, a lo largo de este curso, utilizaremos un ensamblador específico con el fin de poder introducirnos en la mecánica general que envuelve todo el proceso.

Por su facilidad de manejo y su capacidad profesional, el ensamblador elegido es el Ensamblador ULTRAVIOLETA.

Desde un punto de vista práctico, la primera operación a realizar, antes de comenzar a programar en lenguaje de asamblea, es cargar en el Spectrum el programa ensamblador.

El ensamblador se sitúa por encima del RAMTOP, quedando, por tanto, protegido de cualquier eventualidad procedente de las zonas de memoria inferiores.

Con el ensamblador en memoria, podemos comenzar a programar en lenguaje de asamblea, siempre y cuando...

1.º Conozcamos la función que cumple cada nemónico del Z80.

2.º Entendamos las pseudo-operaciones que permite nuestro ensamblador.

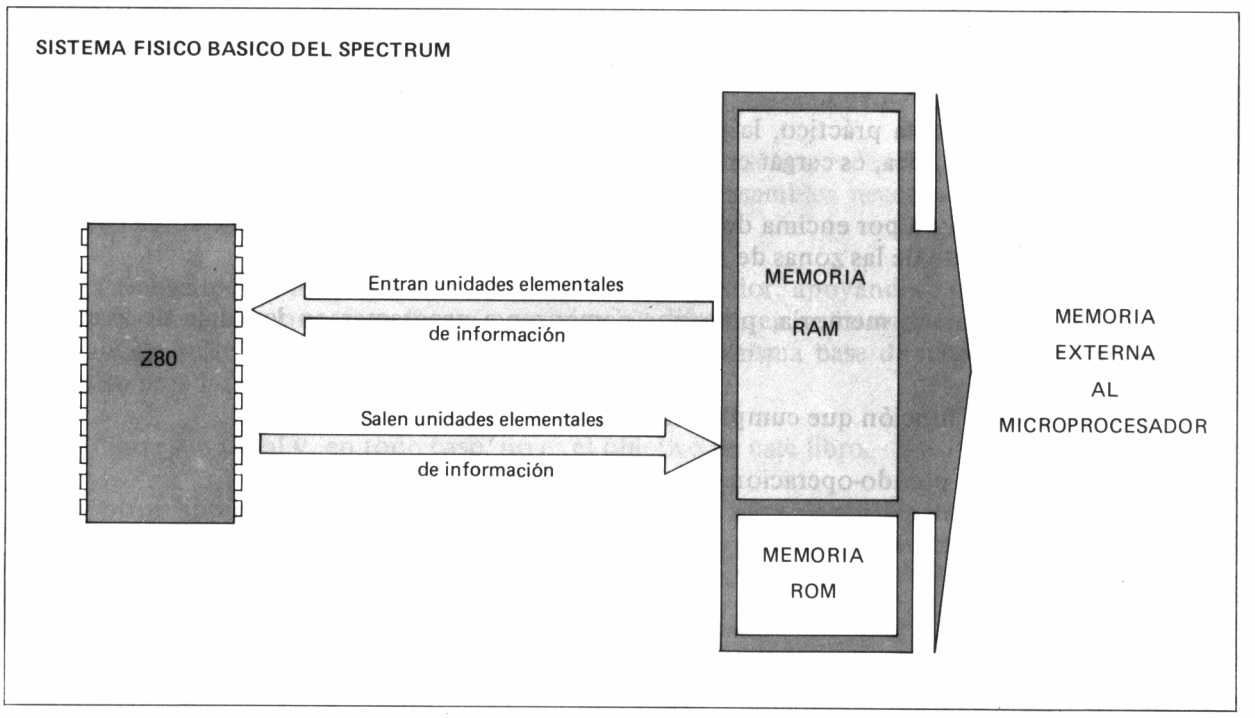
3.º Sepamos manejar las etiquetas.

Microprocesador

En 1972 la compañía INTEL desarrolló un *chip* (circuito integrado) capaz de procesar 8 bits a la vez y, tras una serie de interesantes acontecimientos, dos ingenieros de esta compañía –Faggin y Shima– crean su propia empresa y desarrollan el microprocesador Z80.

En poco tiempo el Z80 se convierte en uno de los más utilizados en microcomputadores y, gracias a este tipo de *chips*, hoy podemos disponer de una máquina tan poderosa como el Spectrum, a precios adecuados a nuestros bolsillos.

De cara a los objetivos propuestos a este libro, la electrónica interna de un microprocesador carece de interés, pero sí debemos tener una primera aproximación al sistema físico básico del Spectrum.



El rectángulo de la izquierda representa el microprocesador Z80 y el de la derecha nos da una idea del mapa de memoria del Spectrum, el cual ya veremos con más detalle posteriormente.

En la realidad la memoria externa del microprocesador está conformada por una serie de chips, cuya única misión es almacenar los bytes representativos de las unidades elementales de información y datos. Así, cuando cargamos en el computador un programa procedente de un soporte externo del tipo casete, microcinta o disco, lo que estamos haciendo realmente es ocupar un cierto espacio de la memoria externa con los *bytes* que conforman el programa.

Con el programa en memoria, una vez que la instrucción oportuna haya sido dada, el microprocesador recibirá –byte a byte– las unidades elementales de información, y de acuerdo con ellas, obedecerá las microinstrucciones y operará sobre los datos, surgiendo, como consecuencia, otras unidades elementales de información que serán enviadas a la memoria para su almacenamiento.

Hasta aquí, una idea de lo que sucede entre el microprocesador y su memoria externa.

Veamos ahora como acontecen las cosas dentro del microprocesador.

Toda la acción se desarrolla entre unos compartimentos denominados *registros.*

En los registros colocaremos datos, los cuales serán procesados conforme a las microinstrucciones.

Conocer estos registros y la función que cumplen es fundamental para escribir un programa en código máquina.

Tenemos un primer contacto con ellos.

REGISTROS DEL MICROPROCESADOR Z80

Registros Principales		Registros Alternativos		Registros de Indice	Puntero de Pila	Contador programa	Vector Interruptor	Refresco Memoria
a	f	a′	f′	ix	sp	pc	i	r
b	c	b′	c′	iy				
d	e	d′	e′					
h	l	h′	l′					

El Z-80, como ya sabemos, es un microprocesador de 8 bits, lo cual significa que es capaz de manejar números binarios de 8 dígitos pero, en función de su diseño, puede procesar también 16 bits.

Así, en el léxico que sigue, llamaremos *registros de 16 bits* a aquellos que, mediante un artificio manejan números binarios de 16 dígitos y *registros de 8 bits* a los que hacen lo propio con 8 dígitos binarios.

Unos comentarios sobre las funciones asignadas a cada registro, completará este tema.

El registro **a** –o acumulador– tiene una función vital y es del tipo 8 bits. Es usado para operaciones artiméticas y comparaciones.

El registro **f** –o de señalizaciones (*flag register*)– sirve para señalar que ciertas condiciones han sido cumplidas dentro de un proceso. Este registro no es de uso directo por el programador.

El resto de los registros principales son también de 8 bits y nos servirán para cualquier trabajo.

Las combinaciones **bc**, **de** y **hl** configuran registros de 16 bits.

Los registros de índice **ix** y **iy**, son de 16 bits.

El registro **iy** no lo puede utilizar el programador.

El puntero de pila **sp**, es de 16 bits y nos indica la dirección de memoria de la parte superior de la pila.

Una vez más: lea y asimile los conceptos sin pretender entenderlos en todos sus extremos. Dé tiempo al tiempo.

El contador de programa **pc**, es de 16 bits e indica al microprocesador en qué dirección de la memoria está la próxima instrucción que debe *leer*.

Los registros **i** y **r** son de 8 bits y no tienen interés, de momento para nosotros.

Con esta perspectiva sobre el microprocesador, pasemos a conocer más de cerca la memoria.

Memoria

Ya sabemos que el microprocesador cumple su función procesando unidades elementales de información, las cuales reclaman de –o almacenan en– su memoria externa.

El microprocesador va reclamando estas unidades de información, byte tras byte, secuencialmente.

En otras palabras, el Z80 leerá, una tras otra y correlativamente, el contenido de las *posiciones de memoria.*

Para poder identificar cada una de las posiciones de memoria posibles, nos referiremos a ellas en función de un número. A este número le denominamos *dirección de memoria.*

Un símil válido para concebir estos dos conceptos, podría ser el de una calle con un vecino por casa y todas las casas en la misma acera. En este caso, cada vecino sería un *byte* –8 bits– o unidad elemental de información.

A cada casa la llamaremos *posición* de memoria y el número que la casa tiene en la calle lo conocemos por *dirección de memoria.*

En este orden de cosas, entenderemos por *zona de memoria,* un grupo de posiciones de memoria cuyas direcciones son correlativas. Algo así como un barrio de casas dentro de la Calle de la Memoria.

El Spectrum sale de fábrica con una zona de memoria ocupada permanentemente, de tal forma que, al inicializar el sistema (al conectar), los primeros 16384 bytes aparecen ya almacenados. Este tipo de memoria se llama **ROM** o memoria a la cual sólo se le puede *reclamar* información.

El resto de la memoria se la denomina RAM o memoria que sirve indistintamente para reclamar o almacenar información.

Pero, incluso dentro de la memoria RAM, existen zonas de memoria que utiliza el sistema para matener su propia organización. Siguiendo con nuestro símil, podríamos decir que la máquina tiene un plan de urbanización para la construcción de casas, las cuales irán siendo habitadas a medida que el programa que estamos introduciendo y el propio Sistema lo requieran.

Estudiemos el mapa de memoria con detalle.

MAPA DE MEMORIA DEL SPECTRUM

VARIABLES DEL SISTEMA	Posiciones DE MEMORIA	DIRECCION DE MEMORIA ACTUAL DE LAS VARIABLES DEL SISTEMA, dado por:	COMENTARIOS SOBRE LA UTILIDAD DE CADA ZONA DE MEMORIA
P-RAMT		PEEK 23732 + 256 • PEEK 23733	Al teclear esta fórmula obtenemos la dirección del último byte de la RAM. En el caso de un Spectrum 48K-65535.
UDG	65368	PEEK 23675 + 256 • PEEK 23676	Con esta fórmula obtenemos la dirección de la posición de memoria donde comienza la zona reservada a los gráficos definibles por el usuario. Tiene 168 bytes más 100 de reserva.
byte delimitador		Código 62 decimal. 3E hexadecimal	Estos bytes delimitan la RAMTOP
byte delimitador		Código Ø.	
RAMTOP		PEEK 23730 + 256 • PEEK 23731	Nos da la dirección de la última posición del sistema. Zona para control de las sentencias GO SUB. Las posiciones de esta zona se llenan de arriba a abajo.
		Puntero de pila (*sp*)	Zona de trabajo reservada al Z80. El puntero de la pila del Z80 siempre señala la última posición rellena.
	bytes disponibles para el usuario		
STKEND		PEEK 23653 + 256 • PEEK 23654	Dirección de la primera posición disponible para el usuario.
STKBOT		PEEK 23651 + 256 • PEEK 23652	Zona utilizada por la pila del calculador. Dirección de la última posición ocupada que será la primera en salir.
			Esta zona de memoria está dedicada a trabajos eventuales y puede ser tan grande o pequeña como requiera.
byte delimitador		Código 13 decimal. D hexadecimal	Este byte delimita la zona de entrada de datos de otras tareas.
WORKSP		PEEK 23649 + 256 • PEEK 23650	Dentro de la zona que arranca con la dirección dada por esta fórmula y hasta el byte delimitador se dedica a la entrada de datos.
E-LINE		PEEK 23641 + 256 • PEEK 23642	Esta zona de memoria está dedicada a la línea de programa BASIC que actualmente se está manejando. Con esta fórmula obtenemos la dirección de arranque de esta zona.
byte delimitador		Código 128 decimal. 80 hexadecimal	Este byte delimita la zona de VARS por arriba.
VARS		PEEK 23627 + 256 • PEEK 23628	Esta zona guarda las variables que utiliza el programa que actualmente esté en memoria y la dirección de la posición de memoria donde comienza viene dada por esta fórmula.
PROG		PEEK 23635 + 256 • PEEK 23636 23755	Esta zona de memoria será tan grande como líneas tenga el programa en BASIC, el cual se almacena con el siguiente criterio: • Los dos primeros bytes definen el número de línea. • Los dos siguientes la longitud de esta línea. • Otro byte para el código de la sentencia BASIC. • Un byte por cada carácter que siga a la sentencia. • Un byte para el código de ENTER. Con esta fórmula obtenemos la dirección de la primera posición de memoria ocupada por un programa BASIC.
byte delimitador		Código 128 decimal 80 hexadecimal	Este byte delimita la zona de CHANS por arriba.
CHANS		PEEK 23631 + 256 • PEEK 23632 23821	El valor obtenido al teclear la fórmula anterior nos indica la dirección de comienzo de esta zona. De tamaño variable, puede llegar a 20 bytes. Controla las rutinas de entrada y salida de información.
ZONA NO VARIABLE		Ø bytes sin microdrives 58 bytes con microdrives	Zona reservada a las variables del Microdrive, si está conectado y en servicio. Ver Apéndice 2 del Manual de Microdrive.
ZONA NO VARIABLE		182 bytes que van desde la posición 23552 a la 23733	Zona reservada a las variables del sistema, las cuales mantienen la organización del mismo. Las posiciones de esta zona son de contenido variable conforme a las necesidades de programación. Ver capítulo 25 del Manual del Spectrum.
ZONA NO VARIABLE		256 bytes que van desde la posición 23296 a la 23551	Zona reservada para el buffer de la impresora (32 caracteres). Al inicializar todas las posiciones contiene Ø.
ZONA NO VARIABLE		768 bytes, que van desde la posición 22528 a la 23295	Zona reservada a los atributos de carácter. Inicialmente todas las posiciones contienen 56 decimales. Este tema se puede ampliar en "Los colores y los gráficos en el Spectrum". ATRIBUTOS
ZONA NO VARIABLE		6144 bytes, que van desde la posición 16384 a la 22527	Zona reservada al fichero de imágenes. El contenido de todas las posiciones de esta zona es Ø al inicializar. Este tema se puede ampliar en "Los colores y los gráficos en el Spectrum". PANTALLA
		16384 bytes de rutinas en código máquina que van desde la posición Ø a la 16383. El aprovechamiento de algunas de estas rutinas se estudia en este libro. Este tema se puede ampliar en "Understanding your SPECTRUM".	MEMORIA ROM Sale grabada de fábrica, de forma indeleble, con el sistema operativo y el intérprete de BASIC.

Conocer y, mejor aún, comprender el Mapa de Memoria de su ordenador –sea este cual sea– es fundamental para obtener el máximo aprovechamiento del mismo. Por esta razón le aconsejo –de cara al Spectrum– que se provea y estudie, en los apartados que le vayan siendo necesarios, los siguientes libros:

* LOS COLORES Y LOS GRAFICOS EN EL SPECTRUM.
* UNDERSTANDING YOUR SPECTRUM
* THE COMPLETE ROM DISASSAMBLEY

Cada uno, por distintas razones, le irán siendo de utilidad, como ya tendremos ocasión de notar.

Todo lo tratado en este epígrafe, no pretenden otra cosa que dar una imagen física de qué es la memoria y cómo se almacena la información, de tal modo que el lector pueda concebir, sin mayor dificultad, las explicaciones que siguen.

Puesta en marcha

Echando un vistazo al Mapa de Memoria esquematizado mas atrás, se puede observar que, en el Spectrum, todo está pensado para trabajar en BASIC.

Esto quiere decir que el programador en código máquina, se las tiene que ingeniar para almacenar su programa de tal forma que no exista peligro de perderlo y, a su vez, buscar la forma de hacerlo funcionar, ya que no existe ninguna sentencia del tipo RUN que "corra" un programa en c.m.

Con respecto a la primera parte de esta problemática, parece evidente que disponemos de una zona específica de almacenamiento, situada a partir de la variable del sistema denominada **STKEND** y hacia arriba, que corresponde a los bytes disponibles.

Aparentemente, tampoco hay inconveniente en almacenar los bytes de nuestro programa de código máquina en la zona de memoria que, comenzando en **PROG**, está destinada, originalmente, para contener un programa en BASIC.

Y, por último, también podríamos utilizar la zona de memoria situada por encima del **RAMTOP**, con lo cual quedaríamos protegidos de cualquier eventualidad procedente de la zona del sistema.

La primera alternativa no es práctica puesto que esta zona de memoria está comprendida entre otras dos zonas de memoria dinámica, lo cual implica que no sabemos cuáles son sus direcciones inicial y final. En otras palabras, desconocemos ciertamente dónde empezar a almacenar los bytes de nuestro programa en código máquina.

Con respecto a la tercera posibilidad –la mejor– tenemos un inconveniente insalvable a la hora de introducir un programa en lenguaje de asamblea que se deriva del hecho de que el programa ensamblador, escrito en código máquina, va a estar situado por encima de la RAMTOP y, por tanto, si nosotros almacenamos los bytes de nuestro programa en esta zona, borraríamos –por superposición– todos, o algunos, de los bytes del ensamblador, a medida que este fuera transcribiendo el lenguaje de asamblea a su equivalente en código máquina.

Así pues, sólo nos queda un lugar de almacenamiento: La zona dedicada al BASIC.

Decidido esto, lo primero que debemos hacer es averiguar cuál es la primera posición de memoria disponible, para ello hay que recordar, previamente, un par de cosas.

En primer lugar, la dirección de memoria donde comienza la zona dedicada al almacenamiento del programa en BASIC, la indica la variable del Sistema **PROG**, la cual viene dada por:

PEEK 23635 + 256 * PEEK 23636

y cuyo resultado es, mientras no usemos microdrives, 23755.

En segundo lugar, sabemos que el comando BASIC *REM* nos permite escribir un grupo de caracteres a continuación del mismo, los cuales no tienen ninguna significación excepto para el pro-

gramador. Por tanto, una línea de programa BASIC perfectamente válida pero sin ningún valor desde el punto de vista del proceso, podría ser:

1 REM 0 0 0 0 0 0 0 0 0 0

Si tecleamos esta línea en el Spectrum, quedaría almacenada en la zona de memoria dedicada al programa en BASIC de la siguiente forma:

En las posiciones 23755 y 23756, el número de línea (el 1).

En las posiciones 23757 y 23758, la longitud de la línea (12). Incluido REM y el ENTER del fin de línea (1 byte por cada uno).

En la posición 23759, el código de la Sentencia REM (234).

En la posición 23760, el código del primer carácter (48), correspondiente al 0.

En las siguientes posiciones tendremos los códigos del resto de los caracteres que siguen a REM.

Visto esto, podremos almacenar los bytes de nuestro programa en código máquina, a partir de la dirección 23760, con lo cual iremos sustituyendo el contenido actual –48 en nuestro ejemplo– de estas posiciones por aquéllos.

Es evidente que, si comenzáramos el almacenamiento en la dirección anterior –la 23759–, borraríamos el comando REM, con lo cual la línea BASIC habría quedado destruida y, consiguientemente, no sería operativa.

Bien, con esta idea sobre el lugar donde –inicialmente– vamos a trabajar, pasamos al terreno práctico.

Coja la cinta que contiene el programa ensamblador y cárgelo, con un LOAD" ", con el que corresponda a su modelo de Spectrum. Cuando aparezca el mensaje final, apriete cualquier tecla y la pantalla quedará como es usual al inicializar el sistema, pero con la diferencia de que por encima del RAMTOP ha quedado posicionado el ensamblador, listo para traducir el programa en lenguaje de asamblea que vayamos a teclear a su equivalente en código máquina.

Hecho esto, ya estamos en condiciones de comenzar a programar en lenguaje de asamblea y la primera línea de cualquier programa en lenguaje de asamblea será, por las razones ya expuestas, una sentencia REM seguida de tantos caracteres como bytes vaya a ocupar el programa una vez traducido a código máquina. Un modelo podría ser:

```
    1 REM 0000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
0000000000000000000000
```

Para su tranquilidad le anticipo que puede considerar la cantidad de bytes holgadamente y, por tanto, el número de caracteres que siguen a la sentencia REM.

No obstante puede considerar que cada instrucción en lenguaje de asamblea requerirá, aproximadamente, un par de bytes en código máquina.

Una vez más, esta instrucción es ficticia ya que sólo sirve para reservar tantas posiciones de memoria como bytes requiera la traducción del programa en lenguaje de asamblea a código máquina.

Con la segunda línea lanzamos el ensamblador, es decir, indicamos al programa ensamblador que comience a traducir instrucciones del lenguaje de asamblea a código máquina.

Y con la tercera línea le decimos al ensamblador a partir de qué dirección debe ir rellenando posiciones de memoria con los bytes resultantes de la traducción que él mismo realiza.

Resumiendo, las tres primeras líneas de un programa en lenguaje de asamblea deben ser:

```
    1 REM 0000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
000000000000000000000
   10 REM go
   20 REM org 23760
```

En este pequeño listado nos han aparecido las dos primeras instrucciones en lenguaje de asamblea: **go** y **org**.

Con la primera lanzamos el ensamblador y con la segunda le indicamos la dirección origen –que como ya sabemos, y mientras no haya microdrives, es la 23760– y todo ello, de acuerdo con las explicaciones dadas más arriba. En el siguiente capítulo ampliaremos un poco más estos conceptos.

Proceso general de trabajo

Ya sabemos cómo y por qué hay que comenzar el programa en lenguaje de asamblea con una sentencia REM seguida de tantos caracteres como bytes se requieran.

También conocemos la dirección más baja posible disponible en la zona de memoria del BASIC y la instrucción **org** imprescindible para que el ensamblador sepa a partir de qué dirección de memoria debe ir colocando el código máquina equivalente.

Conviene anticipar que **go** y **org** no pertenecen al juego de instrucciones del Z80, dicho de otra forma, no son **nemónicos**, son pseudo-operaciones llamadas así porque se asemejan a las instrucciones pero, sin embargo, serán interpretadas exclusivamente por el ensamblador.

Cada ensamblador tiene sus propias **pseudo-op**, muchas de las cuales son comunes a todos ellos, tanto en nombre como en la función que cumplen.

Otra pseudo-op imprescindible en todo programa es **finish**, con la cual indicamos al ensamblador que su función traductora ha concluido. Ejemplo:

```
    1 REM 000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000
   10 REM go
   20 REM org 23760
   30 REM
   40 REM
   50 REM
   60 REM
   70 REM
   80 REM
   90 REM ret
  100 REM finish
```

Entre las líneas 30 y 80 irían las instrucciones de un hipotético programa en lenguaje de asamblea, siguiendo a las sentencias REM, como se verá un poco más tarde. Supuesto esto, sólo tendríamos que añadir las líneas

```
110 RANDOMIZE USR 60000
120 RANDOMIZE USR 23760
```

para completar la estructura de un programa escrito en lenguaje de asamblea y en un Spectrum de 48 K. En un spectrum de 16 K la línea 110 cambiaría a RANDOMIZE USR 27500.

Admitiendo que este programa fuera operativo, todas las líneas serían aceptadas por el *interpreter* del BASIC sin ninguna dificultad y al llegar a la línea 110, en la que nos encontramos con **RANDOMIZE USR** 60000, el control del programa pasa a la subrutina en código máquina situada a partir de la dirección 60000 y que es, precisamente, donde está ubicado nuestro ensamblador.

La instrucción RANDOMIZE USR cumple una función similar a la adjudicada a GO SUB en BASIC, pero con la diferencia de que debe ser utilizada para dirigirse a una subrutina en código máquina.

Una vez que el ensamblador ha sido activado, gracias a RANDOMIZE USR 60000, éste lee todas las posiciones de memoria de la zona del BASIC hasta encontrar **go** y, en ese momento, comienza a traducir microinstrucciones y datos y a colocar los bytes equivalentes en las posiciones de memoria cuya dirección arranca en la indicada por **org**.

El ensamblador, al encontrar la pseudo-op **finish**, *sabe* que el listado del lenguaje de asamblea ha ternimado y devolverá el control al programa en BASIC gracias al último nemónico leído que debe ser **ret**.

En este momento nos encontramos con la línea 120, que obliga a ceder nuevamente el control del programa a otra subrutina en código máquina situada, en esta ocasión, a partir de la dirección 23760.

Es claro que, con esta última instrucción, lo que estamos haciendo es correr nuestra propia rutina en código máquina.

Con estas ideas sobre el proceso general de trabajo, pasaremos a introducirnos en aspectos específicos del lenguaje de asamblea.

Cargando los registros

Páginas atrás vimos que los **registros** son una especie de compartimentos del microprocesador donde se manipulan datos, gracias a las microinstrucciones.

Estos datos hay que introducirlos en los registros y a esta operación la denominamos **carga de registro.**

Para cargar un registro tenemos que recurrir a la microinstrucción definida por el nemónico **ld.**

Un registro lo podemos cargar bien con un dato procedente de la memoria externa o bien de otro registros.

Los datos estarán representados, en todo caso, por bytes ya que el Z80 –como sabemos– es un microprocesador de 8 bits.

En este orden de cosas, es conveniente recordar unos párrafos de "Los Colores y los Gráficos en el Spectrum", libro ya citado y complementario de algunos temas aquí tratados.

"Antes de ir más allá, recordaremos que el número más grande que se puede almacenar en una posición de memoria cualquiera está compuesto por 8 bits –que equivale a un byte– que corresponde a una combinación de ocho "ceros" y "unos". Por tanto, la combinación mínima vendrá dada por 00000000 y la más alta por 11111111, esta representación pertenece al sistema binario. Si lo expresáramos en el sistema decimal diríamos que en una posición de memoria se puede almacenar un número entre 0 y 255.

Así, si usamos una posición de memoria para contar, empezaremos por 0 y seguiremos sin dificultad hasta 255 pero, si aquí intentamos añadir otra unidad, no almacenaremos el número 256 sino que comenzaríamos otra vez a contar desde el cero. Pues bien, para alcanzar números superiores a 255, el computador habilita la siguiente posición de memoria de tal forma que, cada vez que la anterior posición comienza a 0, la nueva aumenta una unidad. En estas condiciones podríamos seguir contando en la primera posición de memoria, mientras la segunda actúa como un contador del número de veces que la primera ha alcanzado 255.

La primera ubicación cuenta números entre 0 y 255 y es conocida como el byte menos significativo (LSB. Least Significant Byte) y la segunda ubicación cuenta de 256 en 256, y es conocida como el byte más significativo (MSB. Most Significant Byte).

Con un PEEK sobre el LSB, obtendremos un número entre 0 y 255 que representa justamente ese número, pero un PEEK sobre el MSB nos dará el número de veces que contiene 256. Esto, en términos que entienda el BASIC y siendo **n** la dirección de memoria del LSB, sería: PEEK n + 256 * PEEK (n + 1), obteniendo de esta forma el número que está almacenado en estas dos posiciones de memoria, el cual, evidentemente, irá desde 0 hasta 65.535.

En sentido contrario podremos trocear un número para almacenarlo en dos ubicaciones de memoria, para ello lo dividiremos previamente entre 256 para saber el número de veces que lo

contiene y almacenarlo en el byte más significativo (MSB) y entonces almacenar el resto en el byte menos significativo (LSB), esto es:

POKE n + 1, INT (N/256)

POKE n, N – 256 * INT (N/256)

Siendo **n** la dirección del LSB y **N** el número a trocear. Con POKE lo que hacemos es colocar los trozos adecuados en las direcciones de memoria seleccionadas."

Retornando al hilo de nuestro tema, diremos que los registros de 8 bits –los **a, b, c, d, e, h, l**– sólo permiten números ENTEROS y POSITIVOS y, claro está, comprendidos entre 0 y 255. El manejo de números negativos y fraccionarios se verá más tarde.

Veamos un ejemplo de carga de un registro.

Supongamos que queremos cargar el registro **c**, con el número 20.

Esto escrito en lenguaje de asamblea sería

INSTRUCCION	*COMENTARIOS*
ld c,20	Cargar el registro **c** con el número 20.

El lenguaje de asamblea, como cualquier otro, tiene una sintaxis –normas de escritura– que viene impuesta por el ensamblador que estemos usando. El nuestro exige, para teclear correctamente la instrucción anterior, escribir **ld**, dejar un espacio, escribir **c** minúscula, una coma y, por último, el número 20 en decimal.

Es importante ser meticulosos con estos detalles ya que, en otro caso, el ensamblador nos obsequiará con su PARAMETER ERROR.

Veamos ahora un ejemplo de carga de un registro con el contenido de otro.

En esta ocasión vamos a cargar el registro **c** con el contenido del registro **a**.

INSTRUCCION	*COMENTARIOS*
ld c,a	Cargar el registro **c** con el contenido del registro **a**.

Esto implica que el byte que estuviera en **c** queda sustituido por el que procede de **a**. El contenido de **a** queda invariable. Ejemplo:

Contenido de los registros antes de **ld c,a**:

Registro **a**	Registro **c**
1 1 1 1 1 1 1 1	0 0 0 0 1 1 1 1

Contenido de los registros después de **ld c,a**:

Registro **a**	Registro **c**
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1

Para cargar una posición de memoria con un byte procedente de un registro, tecleríamos por ejemplo:

INSTRUCCION	*SIGNIFICADO*
ld (32000), a	Cargar la dirección de memoria 32000 con el contenido del registro **a**

Observe que el número 32000 está entre paréntesis, de esta forma nuestro ensamblador lo interpreta como una dirección de memoria.

El par **hl** es usado, generalmente, como un puntero para indicar la dirección de memoria donde está un dato que va a ser cargado a un registro. Note que, en estos casos, el registro **hl** va entre paréntesis. Por ejemplo:

INSTRUCCION	*COMENTARIOS*
ld c, (h l)	Cargar el registro **c** con el contenido de la dirección de memoria indicado por **(h l)**.
ld (h l), c	Cargar la dirección de memoria indicada por **(h l)** con el contenido del registro **c**.

Una aplicación interesante del par de registros **h l**, es que podemos cargar directamente una posición de memoria indicada por **(h l)** con un valor n (entre 0 y 255). Ejemplo:

INSTRUCCION	*COMENTARIOS*
ld (h l), n	Cargar en la dirección de memoria **(h l)** el valor n.

Veamos un ejercicio muy sencillo paso a paso.

Antes que nada cargue el ensamblador en su Spectrum siguiendo las indicaciones ya expuestas.

Supongamos que queremos cargar en el registro **a** el número 15 y a continuación colocar en la dirección de memoria indicada por **h l** el contenido del registro **a**, es decir, el número 15 que previamente le habíamos cargado.

En función de todo lo estudiado, la configuración de nuestro programa será:

Programa	*Comentarios*
1 REM 0 0 0 0 0 0 0 0	Reservamos 8 posiciones de memoria estimadas holgadamente para los bytes que ocupará el programa en C.M. cuando sea traducido por el ensamblador.
10 REM go	Con esta *pseudo-operación* le indicamos al ensamblador **dónde** debe comenzar a traducir los *nemónicos* que siguen y que corresponden a nuestro programa de *lenguaje de asamblea.*
20 REM org 23760	Con esta *pseudo-operación* le indicamos al ensamblador **a partir de qué dirección** de memoria debe ir colocando los bytes procedentes de la traducción del lenguaje de asamblea al C.M.

30 REM ld a, 15	Con esta instrucción de lenguaje de asamblea ordenamos al microprocesador que cargue el registro **a** con el número 15, una vez claro está, que haya sido traducida a C.M. por el ensamblador.
40 REM ld (32000), a	Con esta instrucción en lenguaje de asamblea ordenamos al microprocesador que escriba en la dirección de memoria 32000 el contenido que, a la sazón, tenga el registro **a**, una vez traducida, claro está, a C.M. por el ensamblador.
50 REM ret	Con esta instrucción, una vez traducida a C.M., devolveremos el control al programa BASIC principal. Dadas las características del Spectrum, todos los programas en C.M. son tratados como subrutinas del programa principal que debe ser en BASIC. La forma de indicar que la subrutina en C.M. ha terminado es mediante el nemotécnico **ret**.
60 REM finish	Esta *pseudo-operación* le indica al ensamblador dónde acaba su función traductora de lenguaje de asamblea a C.M.
70 RANDOMIZE USR 60000	Esta instrucción BASIC hace entrar en acción el programa ensamblador que previamente habíamos cargado en el Spectrum, traduciendo del lenguaje de asamblea a C.M. y desde donde aparece la pseudo-instrucción **go** hasta la *finish*, colocando los bytes resultantes a partir de la dirección de memoria indicada por la *org*, y que en este programa es 23760.
80 RANDOMIZE USR 23760	Esta instrucción BASIC, al igual que la anterior, es una llamada a la subrutina en C.M. situada, en este caso, a partir de la dirección 23760. La instrucción **ret** que figura en la subrutina en C.M. devuelve el control a la línea 90 del programa principal.
90 PRINT PEEK 32000	Esta instrucción BASIC nos imprime en pantalla el contenido de la dirección de memoria 32000. En este caso aparecerá en el ángulo superior izquierdo el número 15, que es el que corresponde, según se ordenó con la instrucción ld (32000), a.

Cuando haya tecleado este programa, un *LLIST* –si tiene impresora conectada– le dará:

Ejercicio "1"

```
 1 REM 00000000
10 REM go
20 REM org 23760
30 REM ld a,15
40 REM ld (32000),a
50 REM ret
60 REM finish
70 RANDOMIZE USR 60000
80 RANDOMIZE USR 23760
90 PRINT PEEK 32000
```

Corra el programa con un RUN, ENTER y obtendrá:

```
org 23760
23760 3E 0F          ld a,15
23762 32 00 7D       ld (32000),a
23765 C9             ret
15
```

El 15 que figura en último término del listado anterior corresponde a la instrucción dada al ordenador en la línea 90.

En el caso de que el ensamblado del programa ocupara más de una pantalla, el ensamblador se detendrá cada vez que se llene una pantalla y, en estas circunstancias, tendremos tres opciones:

A. Apretar la tecla **SPACE** para cancelar el proceso de ensamblado.

B. Apretar la tecla **P** para obtener un listado por impresora (o COPY para la última pantalla).

C. Cualquier otra tecla para que continúe el ensamblado.

Esta es la "quinta esencia" de un programa en "código máquina", y éste será el procedimiento básico de trabajo.

A medida que vayamos ampliando nuestro léxico de nemónicos y la forma de usarlos, podremos desarrollar programas más complejos.

¿Subimos otro escalón?

No sufra, no quedan muchos.

Si tiene a mano el manual del Spectrum, ábralo por la página 183 y, en la columna encabezada por *Ensamblador del Z80,* ponga un puntito a la izquierda de los nemónicos que empiezan por **ld** y así podrá ver que ya está capacitado para interpretar muchos de ellos.

Usando los registros

Incrementar y decrementar

Puesto que cargar un registro y transferir información entre éste y la memoria no constituye ya un secreto para nosotros, vamos a ver qué podemos hacer con estos nuevos conocimientos.

Si revisamos el conjunto de nemónicos del Z80 –pág. 183 y siguientes del manual del Spectrum– veremos unas instrucciones que empiezan por **inc** y otras por **dec**, tales como:

Nemónico	*Comentario*
inc bc	Incrementa en 1 el contenido del par de registro **bc**.
inc b	Incrementa en 1 el contenido del registro **b**.
dec b	Decrementa en 1 el contenido del registro **b**.

Y así podríamos seguir con todas las restantes donde intervengan **inc** o **dec**.

Pocas aclaraciones se requieren al margen de los comentarios ya hechos, no obstante un ejemplo fijará la idea de cómo trabajan estas instrucciones y, en todo caso, nos servirá de práctica.

> *Recuerde: Haga todos los experimentos que se le ocurran con los conocimientos que va asimilando. Cuanto más practique mejor.*

Carge el ensamblador en el Spectrum e introduzca el programa del ejercicio anterior, bien tecleándolo otra vez o bien cargándolo de la cinta "Ejercicios" con el nombre "1".

Introduzca ahora la línea

35 REM inc a

Con lo cual el programa quedaría así: Ejercicio "2"

```
   1 REM 00000000000000000000000
00000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld a,15
  35 REM inc a
  40 REM ld (32000),a
  50 REM ret
  60 REM finish
  70 RANDOMIZE USR 60000
  80 RANDOMIZE USR 23760
  90 PRINT PEEK 32000
```

Córralo y obtendremos esta pantalla.

```
org 23760
23760 3E 0F          ld a,15
23762 3C             inc a
23763 32 00 7D       ld (32000),a
23766 C9             ret
16
```

El número 16 es, efectivamente, el número esperado ya que en la línea 35 sumamos uno al contenido del registro **a**.

Introduzca ahora el ejercicio anterior. (En la cinta "Ejercicios" con el nombre "2") o, en todo caso, restituya la línea 1 con un REM seguido de suficientes 0.

¿Ya está?

Coloquemos estas dos líneas nuevas:

36 REM dec a
37 REM dec a

La respuesta que aparece finalmente, claro está, es 14.

Las dos instrucciones anteriores –**inc** y **dec**– no impiden que podamos hacer sumas y restas con valores superiores a 1. Los nemónicos **add** y **sub** cumplen esta función y los estudiaremos en el siguiente capítulo. De momento vaya a la página 183 del manual y ponga otro puntito a la izquierda de estos nuevos nemónicos.

Ya quedan menos.

Antes de seguir, vamos a interpretar el listado del programa anterior ya ensamblado.

La columna de la izquierda nos da las diferentes direcciones de las posiciones de memoria ocupadas por los bytes del programa en C.M.

Los pares de números hexadecimales que figuran a la derecha nos dan el contenido de las posiciones de memoria anteriores y que corresponden al código máquina propiamente dicho.

Por último, a la derecha, tenemos el programa en lenguaje de asamblea.

Observe, finalmente, que hay tantas direcciones de memoria en la columna de la izquierda como pares de dígitos hexadecimales.

Sumando y restando

En la página 187 del manual de Spectrum, y a la altura del código 198, Vd. encontrará el nemónico **add a, N**, que nos viene a decir "añade al registro **a** el número **N**".

Cargue el *ensamblador*

El siguiente ejemplo está en la cinta "ejercicios" con el nombre "3". Cárguelo o, si no tiene la cinta, teclee lo que sigue.

Programa

```
   1 REM 00000000000000000000000
00000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld a,15
  40 REM add a,100;ld (30000),a
  50 REM ret
  60 REM finish
  70 RANDOMIZE USR 60000
  80 RANDOMIZE USR 23760
 100 PRINT PEEK 30000
```

```
org 23760
23760 3E 0F        ld a,15
23762 C6 64        add a,100
23764 32 30 75     ld (30000),a
23767 C9           ret
115
```

Comentarios

Observe que en la línea 40 hemos ordenado que se sume el número 100 al contenido del registro *a* y, separado por un punto y coma (;), hemos añadido otra instrucción. A las líneas de este tipo las llamaremos multisentencia.

y la respuesta final obtenida, al ejercutar este programa, es la esperada: 115.

Ponga otro puntito en la instrucción **add a, N**, por favor.

Teclee NEW, ENTER.

¿Qué tendríamos que hacer si quisiéramos sumar al registro **a** el contenido de cualquier otro registro de 8 bits?

En primer lugar echemos un vistazo a la página 186 del manual y veremos, entre los códigos 128 y 135, un conjunto de instrucciones del tipo **add a, b.**

En todos figura el registro **a** en primer lugar, lo cual no es extraño si consideramos que es el *acumulador*. Este tipo de nemónico nos viene a decir *acumula el contenido del registro que figura en segunda posición.*

Fíjese que en esa segunda posición puede estar el propio registro *a* con lo cual se duplicaría el contenido del registro **a** (o acumulador).

Como siempre, un ejemplo será de gran ayuda.

¿Tiene el ensamblador en memoria?

Carge o teclee el ejercicio "4".

Programa

```
   1 REM 00000000000000000000000
00000000000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld c,20
  40 REM ld a,100;add a,c
  50 REM add a,a
  60 REM ld (32000),a
  70 REM ret
  80 REM finish
  90 RANDOMIZE USR 60000
 100 RANDOMIZE USR 23760
 110 PRINT PEEK 32000
```

```
org 23760
23760 0E 14          ld c,20
23762 3E 64          ld a,100
23764 81             add a,c
23765 87             add a,a
23766 32 00 7D       ld (32000),a
23769 C9             ret
240
```

Comentarios

En la segunda sentencia de la línea 40 añadimos al registro *a* el contenido del registro *c* y en la línea 50 añadimos al registro *a* su propio contenido.
En la línea 110 pedimos el contenido de la dirección 32000, que debería ser 240.

Corra el programa y compruebe el resultado con el esperado.

Habrá observado que en algunos nemónicos del tipo **add** aparece el par de registros **hl** de 16 bits. En unos tenemos **add a, (hl)**, en otros **add hl, bc**, o **add hl, de** o **add hl,hl**.

Un poco más adelante veremos un ejemplo sobre estas instrucciones.

Como se indicó en el capítulo CARGANDO LOS REGISTROS, cuando el par de registro **hl** aparece entre paréntesis –**(hl)**– lo estamos usando como un puntero que nos dirije a una posición de memoria, por tanto la instrucción **add a, (hl)**, la interpreta el ensamblador como *sumar al registro* a *el contenido de la posición de memoria que señala* **(hl)**.

Cargue el ejercicio "5" o teclee:

Programa

```
   1 REM 00000000000000000000000
00000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld a,10
  40 REM ld hl,30000
  50 REM ld (hl),50
  60 REM add a,(hl)
  70 REM ld (32000),a
  80 REM ret
  90 REM finish
 100 RANDOMIZE USR 60000
 110 RANDOMIZE USR 23760
 120 PRINT PEEK 32000
```

```
org 23760
23760 3E 0A          ld a,10
23762 21 30 75       ld hl,30000
```

Comentarios

En la línea 40 cargamos el registro **hl** con el número 30000, en la línea 50 cargamos la dirección dada por **(hl)** –que es 30000– con el número 50.
En la línea 60 añadimos al contenido actual de **a** –que es 10– el contenido de la dirección de memoria que señala **(hl)**.
El resultado que la línea 120 nos debería dar es 60.

```
23765 36 32          ld (hl),50
23767 86             add a,(hl)
23768 32 00 7D       ld (32000),a
23771 C9             ret
60
```

y, al ensamblar, aparecerá 60, confirmando nuestro trabajo.

El registro **a** se denomina *acumulador* porque está específicamente habilitado –como ya se apuntó– para todas las labores aritméticas, pero es además un registro de 8 bits. Esto significa que no puede usarse para sumar números que exijan 16 bits.

Para superar esta limitación el microprocesador Z80 nos permite usar el par **hl** como *acumulador* para sumas de números de 16 bits.

Veamos el siguiente ejemplo "6":

Programa

```
   1 REM 00000000000000000000000
0000000000
  10 REM go
  20 REM org 23760
  30 REM ld hl,41000
  40 REM ld bc,1000
  50 REM add hl,bc
  60 REM ld (32000),hl
  70 REM ret
  80 REM finish
  90 RANDOMIZE USR 60000
 100 RANDOMIZE USR 23760
 110 PRINT PEEK 32000+256*PEEK 3
2001
```

Comentarios

En las líneas 30 y 40 cargamos los registros **hl** y **bc** con números superiores a 255, mayores, por tanto, a 8 bits en binario. En la línea 50 añadimos al contenido del registro **hl**, el del registro **bc**.
Observe que en la línea 110 tenemos que utilizar el contenido del byte 32000 y el del byte 32001.

Primera pantalla de este ejercicio:

```
org 23760
23760 21 28 A0       ld hl,41000
23763 01 E8 03       ld bc,1000
23766 09             add hl,bc
23767 22 00 7D       ld (32000),hl
23770 C9             ret
42000
```

El resultado esperado es 42000 . Corra el programa y compruébelo.

Si ha comprendido la interpretación de estos nemónicos, ya está en condiciones de poner los puntitos correspondientes a la izquierda de los mismos.

Y, ¿qué hay de la resta?

Pues la sustracción es, hasta cierto punto, más sencilla ya que sólo puede utilizar el acumulador –*registro a*– para trabajar. Esto quiere decir que allá donde veamos la instrucción **sub** seguida de un número –por supuesto entre ϕ y 255– se producirá una sustracción entre el contenido del regitro **a** y el número indicado en la sentencia **sub**.

Ejercicio "7":

Programa

```
  1 REM 00000000000000000000000
 10 REM go
 20 REM org 23760
```

Comentarios

En la línea 40 restamos el número 3 del contenido del registro *a*, que es 15 según la

```
 30 REM ld a,15
 40 REM sub 3
 50 REM ld (32000),a
 60 REM ret
 70 REM finish
 80 RANDOMIZE USR 60000
 90 RANDOMIZE USR 23760
100 PRINT PEEK 32000
```

línea 30.
El resultado esperado es 12.

```
org 23760
23760 3E 0F        ld a,15
23762 D6 03        sub 3
23764 32 00 7D     ld (32000),a
23767 C9           ret
12
```

y el contenido en la dirección 32000 es, claro está, 12.

Si en lugar de querer restar un número determinado del contenido del registro *a*, quisiéramos restar el contenido de cualquier otro registro del tipo 8 bits, bastaría con escribir **sub** seguido del nombre que identifique a ese registro. Ejemplo: **sub c**.

Con estos nemónicos ya estaríamos en condiciones de pensar en operaciones aritméticas más complicadas. Ya veremos esto, pero antes vamos a estudiar cómo hacer cambiar al microprocesador su secuencia de lectura normal.

Saltando

Supongamos que hemos tecleado el siguiente programa, con el **ensamblador** previamente cargado.

Cargado con el "8" en la cinta de *Ejercicios.*

```
   1 REM 00000000000000000000000
00000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld hl,32000
  40 REM ld c,0
  50 REM Aqui;ld a,c
  60 REM add a,2
  70 REM inc hl
  80 REM ld c,a
  90 REM ld (hl),c
 100 REM ld a,20
 110 REM cp c
 120 REM jp nz,Aqui
 130 REM ret
 140 REM finish
 150 RANDOMIZE USR 60000
 160 RANDOMIZE USR 23760
 170 FOR x=32000 TO 32011
 180 PRINT PEEK x
 190 NEXT x
 200 STOP
```

Observará que en las líneas 50, 110 y 120 hemos introducido algunas instrucciones cuyos significados desconocemos.

Analicemos, línea a línea, la interpretación de este programa.

Programa	*Comentarios*
1 REM 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Como ya vimos, con esta introducción reservamos 40 bytes para el programa en C.M. 1 bytes por cada carácter que sigue a la sentencia REM.
10 REM go	Aquí lanzamos el *ensamblador* a traducir del lenguaje de asamblea a C.M.
20 REM org 23760	Esta *pseudo-instrucción* indica al *ensamblador* a partir de qué dirección de memoria debe ir colocando los bytes procedentes de la traducción del lenguaje de asamblea al C.M.
30 REM ld hl, 32000	Este *nemónico* carga el par de registros **hl** con el número 32000.

40 REM ld c, 0	Cargamos el registro **c** con el número ϕ.
50 REM Aqui; ld a, c	*Atención:* La palabra **Aquí** es una *etiqueta* (label) que nos servirá para dirigirnos a **Aquí** desde cualquier lugar del programa, como ya veremos. Con el particular *ensamblador* que en este curso utilizamos, las etiquetas deben comenzar con una letra mayúscula y pueden estar conformadas por tantos caracteres como el programador desee. En otros *ensambladores* se puede exigir un máximo de 6 caracteres, o cualquier otra condición. Inmediatamente a continuación de la etiqueta aparece un punto y coma (;). En este caso, después del punto y coma, se ordena al microprocesador que cargue el registro **a** con el contenido del registro **c**.
60 REM add a, 2	Añadimos al contenido del registro *a* el número 2.
70 REM inc hl	Ahora incrementamos el contenido del registro **hl** en una unidad.
80 REM ld c, a	Cargamos el registro *c* con el contenido del registro *a*.
90 REM ld (hl), c	Como recordará, al estar el registro **hl** entre paréntesis, estamos indicando al microprocesador que cargue la dirección de memoria señalada por el número contenido en el registro **hl**, con el contenido del registro **c**.
100 REM ld a, 20	Se carga el registro **a** con el número 20.
110 REM cpc	*Atención:* La instrucción **cp** (*comparar*) seguida de un registro —en este caso **c**— compara el contenido del registro **a** con el contenido del registro referido. Esta instrucción la estudiaremos en todas sus posibilidades más adelante en este capítulo. Específicamente, en este programa, en la línea 100 hemos cargado el registro **a** con el número *20*, esto quiere decir que, debido a la instrucción **cp c**, se va a comparar el contenido del registro **c**, con el número 20 que es el contenido del registro **a**.
120 REM jp nz, Aqui	*Atención:* La instrucción **jp** obliga a un salto a una dirección de memoria. La instrucción **jp nz**, en el caso específico de este pro-

grama, obliga al microprocesador a dirigirse a la dirección de memoria donde comienza la etiqueta **Aquí**, sólo en el caso de que el resultado de la comparación realizada en la línea anterior (**cp c**) sea distinto de cero (No Zero).
Si fuera ϕ el resultado de la comparación, el microprocesador no salta a la etiqueta y sigue su secuencia normal de lectura, es decir, pasa a la siguiente dirección de memoria.

130 REM ret

ret devuelve el control al programa principal del que hubiera salido, en este caso al programa en BASIC que figura más abajo. La instrucción **ret**, como ya se verá, también puede retornar –bajo ciertas condiciones– al programa principal en C.M.

140 REM finish

Esta pseudo-instrucción indica al *ensamblador* que su función traductora ha terminado.

150 RANDOMIZE USR 60000

Cuando el ordenador llega a esta línea BASIC, se inicia efectivamente el proceso de traducción del lenguaje de asamblea al C.M., desde donde figura ***go*** hasta donde aparece *finish*.

160 RANDOMIZE USR 23760

Una vez realizado el trabajo anterior, y gracias a esta instrucción, se llama a la subrutina en C.M. que comienza en la dirección 23760. Cuando esta subrutina encuentra la instrucción **ret** en su proceso de lectura, devuelve el control a la siguiente instrucción BASIC, que en nuestro caso está en la línea 170.

Las líneas 170, 180 y 190 simplemente obligan a imprimir el contenido de las posiciones de memoria desde la dirección 32000 hasta la dirección 32011.

Este sería el resultado de ejecutar el programa anterior:

```
org 23760
23760 21 00 7D      ld hl,32000
23763 0E 00         ld c,0
Aqui
23765 79            ld a,c
23766 C6 02         add a,2
23768 23            inc hl
23769 4F            ld c,a
23770 71            ld (hl),c
23771 3E 14         ld a,20
23773 B9            cp c
23774 C2 D5 5C      jp nz,Aqui
23777 C9            ret
```

```
16
2
4
6
8
10
12
14
16
18
20
0
```

Contenido	*Direccion memoria*	*Comentario*
0	32000	Por la configuración del programa debe ser ϕ.
2	32001	
4	32002	
6	32003	
8	32004	Cada posición de memoria aumenta en 2 el contenido
10	32005	del registro **c** que a su vez hemos cargado en la direc-
12	32006	ción (**hl**), la cual aumenta en cada bucle en 1.
14	32007	
16	32008	
18	32009	
20	32010	
0	32011	Por la configuración del programa debe ser ϕ.

Un comentario general sobre el programa aclarará cualquier duda que pueda quedar.

En las líneas 30 y 40 iniciábamos los registros **hl** y **c**, los cuales sólo serán leídos una vez ya que la *etiqueta* está situada posteriormente.

En la línea 50 se sitúa una *etiqueta* que, en este caso y por nuestra voluntad, hemos denominado *Aquí*. También cargamos el registro **a** con el contenido *actual* del registro **c**. En el primer bucle este valor será 0.

En la línea 60 sumamos 2 al contenido *actual* del registro *a*. En el primer bucle el resultado de esta instrucción será 2.

En la línea 70 incrementamos en 1 el contenido del registro **hl**. En el primer bucle el contenido de **hl** pasa de 32000 a 32001. Por esta razón un PEEK 32000 da 0.

La línea 80 carga el registro **c** con el valor *actual* de **a**. En el primer bucle este valor será 2, como ya hemos dicho unas líneas más arriba.

En la línea 90 cargamos la posición de memoria cuya dirección está definida por (**hl**), con el valor actual de **c**. En el primer bucle, **hl** contiene el número 32001 y en cada bucle aumenta en una unidad.

En la línea 100, cargamos el registro **a** con un nuevo valor –el 20– que nos servirá de modelo para la comparación posterior.

En la línea 110 se produce una comparación –**cp**– entre el contenido *actual* del registro **a** –20– y el del valor *actual* del registro **c** (que dará sucesivamente 2, 4, 6, ..., 20). El resultado de esta comparación será distinto de 0, hasta que **c** alcance el valor 20.

En la línea 120 se obliga al microprocesador a dirigirse a la etiqueta *Aquí*, mientras el resultado de la comparación anterior sea distinto de 0. Cuando la comparación anterior dé como resultado 0, entonces la lectura sigue.

A partir de la línea 130 ya conocemos el significado de las instrucciones y la intención del resto del programa es clara.

Con el ejemplo anterior hemos visto un aspecto de las aplicaciones posibles de las instrucciones **cp**, **jp** y de la utilidad de las etiquetas.

Estudiemos ahora todas las alternativas que nos ofrecen y algunos temas implicados.

Hasta el momento hemos desarrollado programas en lenguaje de asamblea sin habernos parado a conocer las reglas fundamentales de la escritura –o sintaxis– que rigen esta forma de programación. Algunas de ellas ya estarán intuidas por el lector, no obstante, las definiremos explícitamente.

Cada línea de un programa en lenguaje de asamblea debe responder a esta configuración:

campo 1 campo 2 campo 3 campo 4

El campo 1 es opcional y, consiguientemente, no tiene por qué aparecer en todas las líneas.

Destinado a contener las *etiquetas* (labels), debe estar separado del *campo 2* por un *delimitador* que, en el caso de nuestro *ensamblador*, es un punto y coma (;) en otros puede ser un espacio en blanco. La sintaxis específica de nuestro *ensamblador*, exige que la primera letra del nombre de la etiqueta sea mayúscula y no tiene límite en cuanto al número de caracteres que componen el nombre de la misma.

El nombre de la etiqueta identifica una línea del programa en lenguaje de asamblea, pudiéndonos dirigir a ella desde cualquier parte del programa con la única condición de que no haya más que una etiqueta con ese nombre entre la llamada y la propia etiqueta.

Como hemos podido apreciar en el ejemplo introductorio de este capítulo, las etiquetas son sumamente útiles para dirigirnos a subprogramas y direccionar los saltos (**jp**).

El *campo 2* está reservado para los *nemónicos* –o, alternativamente, para las pseudo-instrucciones– en la forma ya utilizada en los ejemplos anteriores.

Todos los nemónicos están compuestos por algunas letras de la palabra inglesa que identifica la operación que tiene encomendada. Ejemplo:

ld ... load ... cargar

Cada nemónico tiene un código que Vd. puede encontrar en el **Apéndice A** del manual de su Spectrum. Ejemplo:

ld a,N ... 62 decimal ... 3E hexadecimal

El *campo 3* sitúa los *operandos*.

Los operandos pueden ser *únicos* o *dobles*.

Un ejemplo de operando único podría ser:

campo 2	campo 3
Sub	**10**

Un ejemplo de operando doble podría ser:

campo 2	campo 3
ld	**a,10**

Obsérvese que los dos operandos del ejemplo anterior van separados por una coma (,). Esta regla sintáctica no debe olvidarse.

El operando situado a la izquierda de la coma, o primer operando, es el *operando destino*, el segundo, a la derecha de la coma, es el *operando origen*.

Como ya sabemos, *un operando entre paréntesis se refiere siempre al contenido de la posición de memoria cuya dirección está especificada por el operando entre paréntesis.*

El *campo 4* se utiliza opcionalmente para comentarios del programador con el fin de aumentar la claridad del programa en lenguaje de asamblea.

La sintaxis de este campo exige estar separado del campo 3 por un punto y coma (;) y comenzar el comentario propiamente por un signo de admiración (!).

Con esto en nuestro haber pasemos a analizar los nemónicos **cp** y **jp**.

Supongamos una secuencia de direcciones de memoria cuyas posiciones están ocupadas por imaginarios bytes de un programa en C.M., en el cual tenemos una instrucción de *salto incondicional* a una dirección de memoria situada antes de la posición que ocupa esta instrucción de salto.

Las instrucciones de *salto incondicional* obligan al microprocesador a seguir la lectura del programa en C.M. en la dirección de memoria que sigue al nemónico representativo del salto, así:

jp 32000

Obliga a seguir la lectura a partir de la dirección 32000.

También es posible este otro ejemplo:

jp Aquí

Con lo cual el programa sigue a partir de la etiqueta **Aquí**.

Hecha esta salvedad, seguimos con el imaginario programa en C.M. propuesto más arriba.

Esquemáticamente ésta podría ser la secuencia de direcciones de memoria:

Direcciones memoria	Contenido memoria	
23760	00010101	
23761	00000000	
23762	01111101	
23763	00001110	
23764	11000011	Estos tres bytes equivalen a la instrucción en lenguaje de asamblea *jp 23760*, una vez traducido por el *assembler*.
23765	11010000	
23766	01011100	
23767	00010101	
23768	00000000	
23769	01111000	

El recorrido de lectura del programa sería el indicado por las flechas del esquema anterior. Comenzamos la lectura en la dirección 23760 y al llegar a la dirección 23766, final de la instrucción **jp 23760**, volvemos a la 23760. Hemos generado un bucle sin fin.

Si por el contrario hubiéramos saltado incondicionalmente a la dirección 23769, p.e., lo único que hubiéramos conseguido sería saltarnos un par de posiciones de memoria, que es lo mismo que no haberlas escrito.

En otras palabras, los saltos incondicionales deben ser tratados con cuidado por el programador, ya que si genera un bucle sin fin, sólo podrá salir de él desconectado el Spectrum.

Entendemos por *salto condicionado* aquél que se da en la lectura de un programa, sólo cuando se cumplen ciertas condiciones.

Recordemos en este punto, las líneas 110 y 120 del programa con que se abrió este capítulo:

110 REM cp c
120 REM jp nz, Aquí

Decíamos, comentando estas líneas, que el nemónico **cp** comparaba el operando **c** con el contenido del registro **a**, añadimos ahora que, tanto el registro **a** como el **c**, no varían su contenido después de la comparación, esto implica, evidentemente, que el resultado de la comparación ha quedado guardado en algún lugar dentro del microporcesador, ya que en caso contrario la instrucción **jp nz, Aquí** –salto a la etiqueta Aquí en el supuesto de que la comparación no sea 0– no sabría cuál ha sido el resultado de la comparación y, por tanto, no podría actuar.

Efectivamente, el registro **f** de señalizaciones –*flag register*– cumple esta función entre otras.

Una ojeada parcial a este registro nos mostrará lo sucedido con las líneas 110 y 120 origen de este comentario.

El registro **f** es, fundamentalmente, un grupo de bits cuya misión individual es indicar –señalizar– si la última operación, aritmética o lógica, cumple o no determinada condición .

La configuración de este registro responde al esquema

Registro **f**	7	6	5	4	3	2	1	0	posición del bit
	s	z		h		p/v	n	c	nombre del bit

De todos los bits que componen este registro, vamos a estudiar el **z** y el **s** en esta ocasión, ya que el primero está directamente implicado en nuestra actual discusión y el **s** será fácil de entender.

El bit de señalización denominado **z**, tiene por misión indicar si la última operación efectuada por el microprocesador en el registro **a**, fue cero o distinta de cero.

Para ello el bit **z** contendrá el dígito 0 indicando, de este modo, que el resultado de la comparación fue **no cero** y contendrá el dígito 1 si este resultado, por el contrario, fue **cero**.

En función de esto y volviendo a las líneas:

110 REM cp c 120 REM jp nz, Aquí

tendremos que el resultado de la comparación de la línea 110, podrá ser **cero** o **no cero**, y, consiguientemente, el *flag* **z** contendrá 1 ó 0.

Cuando el microprocesador pasa a la instrucción de la línea 120, "mira" en el bit **z** del registro **f** y si este bit es 0, interpreta que la comparación fue **no cero** y dirige la lectura del programa a la etiqueta **Aquí**. En caso contrario se dirige al byte siguiente, ignorando la etiqueta.

Hasta aquí la función del *flag* **z** del registro **f**. Con respecto al *flag* **s** del mismo registro, su función es similar al anterior pero en vez de guardar ϕ o 1 según sea **no cero** o **cero** el resultado de la comparación, aquí guardaremos un 1 si el resultado de la última operación produce un valor negativo y un ϕ si el resultado es positivo.

Esquema:

bandera S

Resultado última operación	Contenido del bits
Negativo (−)	1
Positivo (+)	ϕ

Vamos a acabar este capítulo interpretando algunas instrucciones de salto condicionadas al contenido de los *flags* **z** y **s**.

Instrucción	*Comentario*
jp nz, Aquí	Esta instrucción nos indica, como ya hemos visto, que la lectura del programa seguirá en la etiqueta **Aquí** si el resultado de la comparación fue distinto de cero.
jp z, Aquí	En esta ocasión, por el contrario, la lectura seguirá en la etiqueta **Aquí** si el resultado de la comparación fue cero.
jp p, Aquí	El programa seguirá en la etiqueta **Aquí** sólo si el resultado de la comparación fue positivo.
jp m, Aquí	El programa seguiría en la etiqueta **Aquí** sólo si el resultado de la comparación fue negativo.

En el siguiente capítulo un nuevo y muy interesante tipo de salto... a las subrutinas.

Subrutinas

Con las instrucciones de salto estudiadas anteriormente, cambiábamos la secuencia de lectura de un programa por el microprocesador sin posibilidad de volver a la instrucción siguiente al nemónico **jp** que originó el salto.

Las subrutinas nos permiten, siguiendo ciertas normas que veremos a continuación, saltar a una dirección de memoria distinta a la secuencia normal y volver al programa principal una vez leída una serie de instrucciones, de forma similar al *modus operandi* que sigue la sentencia GOSUB del lenguaje BASIC.

Cuando en un programa aparece el nemónico **call**, seguido normalmente de una etiqueta, se está indicando al microprocesador que salte a esa etiqueta y que siga la lectura del programa a partir de ella y hasta encontrar una instrucción **ret**, en ese momento la lectura del programa continúa en la instrucción siguiente al **call** que originó el salto.

Para ver estos nemónicos trabajando de una forma práctica, confeccionemos un programa que obtenga los resultados de la tabla de multiplicar por 5.

Ejercicio "9"

```
   1 REM 000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld hl,32000
  40 REM ld a,0
  50 REM ld (hl),0
  60 REM call Suma
  70 REM call Suma
  80 REM call Suma
  90 REM call Suma
 100 REM call Suma
 110 REM call Suma
 120 REM call Suma
 130 REM call Suma
 140 REM call Suma
 150 REM call Suma
 160 REM ret;!fin programa princ
ipal
 170 REM Suma;inc hl;add a,5
 180 REM ld (hl),a
 190 REM ret;!retorno subrrutina
 200 REM finish
 250 RANDOMIZE USR 60000
 260 RANDOMIZE USR 23760
 270 FOR x=32000 TO 32010
 280 PRINT PEEK x
 290 NEXT x
```

Atención:
Vigile que haya suficientes "ceros" tras el primer REM. ¿Los hay en este listado? Pruebe a ejecutar este listado o el que figura en el casete de ejercicios.

Al desarrollar un programa, el programador se encuentra con que ciertos procesos se repiten, teniendo que optar por escribirlos una y otra vez o, por el contrario, generar una subrutina a la cual se recurrirá tantas veces como sea necesario. Esto último será lo más práctico.

También, un buen criterio exigirá crear tantas subrutinas como la claridad del programa pudiera aconsejar. En el caso anterior, cada vez que dirigimos el programa a la subrutina etiquetada bajo el nombre *Suma*, aumentamos en 5 el contenido del registro *a*.

Siendo el resultado final:

```
org 23760
23760 21 00 7D      ld hl,32000
23763 3E 00         ld a,0
23765 36 00         ld (hl),0
23767 CD D7 5C      call Suma
23770 CD DA 5C      call Suma
23773 CD DD 5C      call Suma
23776 CD E0 5C      call Suma
23779 CD E3 5C      call Suma
23782 CD E6 5C      call Suma
23785 CD E9 5C      call Suma
23788 CD EC 5C      call Suma
23791 CD EF 5C      call Suma
23794 CD F2 5C      call Suma
23797 C9            ret
fin programa principal
Suma
23798 23            inc hl
23799 C6 05         add a,5
23801 77            ld (hl),a
23802 C9            ret
retorno subrrutina
0
5
10
15
20
25
30
35
40
45
50
```

La técnica de utilización de subrutinas es la aquí expuesta, no obstante se nos ofrecen diferentes posibilidades a la hora de recurrir a las mismas.

En el ejemplo anterior hemos dirigido *incondicionalmente* la lectura del programa a la subrutina. Esto no quiere decir que sea la única forma.

Analicemos el siguiente programa detenidamente.

Ejercicio "10":

Combinando BASIC y lenguaje de asamblea, desarrollar un programa que permita introducir por teclado un número —menor que 255 y mayor que 27— y, si es menor que 155, se le sumará 5, en caso contrario se mantendrá invariable. Obtener el resultado en pantalla.

Programa

```
   1 REM 00000000000000000000000
00000000000000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld hl,30000;ld e,(hl)
  40 REM ld a,155
  50 REM cp e
  60 REM call p,Suma
  80 REM ret;!fin programa princ
ipal
  90 REM Suma;ld a,e;add a,5
 100 REM ld (hl),a
 110 REM ret;!retorno subrutina
 115 REM finish
 120 RANDOMIZE USR 60000
 130 INPUT "numero",n
 140 IF n>255 OR n<28 THEN GO TO
 130
```

Comentarios

A partir de la línea 130 aparece el programa BASIC, gracias al cual permitimos la entrada por teclado de un número menor que 255 ó mayor que 27.
En la línea 160 hacemos una llamada a la rutina en código máquina que comienza en el byte 23760.
En 170 pedimos la impresión del contenido del byte 30000.

```
150 POKE 30000,n
160 RANDOMIZE USR 23760
170 PRINT PEEK 30000
180 GO TO 130

org 23760
23760 21 30 75      ld hl,30000
23763 5E            ld e,(hl)
23764 3E 9B         ld a,155
23766 BB            cp e
23767 F4 DB 5C      call p,Suma
23770 C9            ret
fin programa principal
Suma
23771 7B            ld a,e
23772 C6 05         add a,5
23774 77            ld (hl),a
23775 C9            ret
retorno subrutina
```

El listado en lenguaje de asamblea comprendido entre la línea 1 y la 115 es fácilmente interpretable por el lector, excepto la línea 60 donde aparece:

60 REM call p, Suma

Esta instrucción condiciona el salto a la subrutina de acuerdo con el resultado de la comparación que se efectúa en la línea anterior. Este sistema de condicionar el salto a la situación en que se encuentre un determinado *flag* ya se discutió en el capítulo *SALTANDO.*

En cualquier caso la interpretación de la línea 60 viene a ser: *Salta a la subrutina denominada Suma si el resultado de la comparación anterior es positivo.* Dicho de otro modo, si el resultado de restar del contenido del registro **a** el contenido del registro **e** es positivo, se salta a la subrutina *Suma.*

Compruebe esto corriendo el programa anterior e introduzca datos que sean claramente significativos, como por ejemplo 200.

Al teclear el número 200, éste queda en la dirección 30000 (línea 150) y posteriormente pasa al registro **e** (línea 30). En las líneas 40 y 50 se efectúa la comparación del contenido de **e** con el de **a** (155), siendo negativo el resultado, lo cual obliga a ignorar el salto a la subrutina **Suma**, siendo el resultado final 200.

Si tecleamos 100, el resultado de la comparación es positivo con lo cual se salta a la subrutina y el resultado sería 105.

En función de las banderas del registro **f** visto hasta ahora, podríamos resumir las condiciones de salto a la subrutina de la siguiente forma

CONDICIONES DE SALTO A LA SUBRUTINA (banderas z y s)

Forma de la instrucción	*Interpretación*
call p, *Nombre etiqueta*	Salto a la subrutina denominada *Nombre etiqueta* si el resultado de la comparación es +.
call m, *Nombre etiqueta*	Salto a la subrutina denominada *Nombre etiqueta* si el resultado de la comparación previa es –.
call z, *Nombre etiqueta*	Salto a la subrutina denominada *Nombre etiqueta* si el resultado de la comparación previa es **0**.
call nz, *Nombre etiqueta*	Salto a la subrutina denominada *Nombre etiqueta* si el resultado de la comparación es **NO 0**.

Existen otras condiciones de salto a subrutinas en función de otras banderas del registro **f** como ya tendrá oportunidad de ver.

Para afianzar estos conceptos observe que, si en el programa anterior cambiamos la condición de salto **p** por **m**, los resultados obtenidos se invertirían.

Cambie la línea *60 REM call p, Suma* por *60 REM call m, Suma* y vuelva a introducir los datos 200 y 100, obteniéndose en esta ocasión como resultados 205 y 100.

Haga algunos ejercicios con **z** y **nz**.

En la misma forma que podemos imponer condiciones al salto a una subrutina, también existe posibilidad de condicionar el retorno de la subrutina. Hasta ahora hemos trabajado con *ret*orno *incondicional.*

CONDICIONES DE RETORNO DE UNA SUBRUTINA (banderas z y s)

Forma de instrucción	*Interpretación*
ret p	Retorno de la subrutina al programa principal si el resultado de la comparación previa es +.
ret m	Retorno de la subrutina al programa principal si el resultado de la comparación previa es −.
ret z	Retorno de la subrutina al programa principal si el resultado de la comparación previa es 0.
ret nz	Retorno de la subrutina al programa principal si el resultado de la comparación previa es **NO** 0.

El ejercicio que sigue nos servirá de muestra para ver en acción las condiciones de retorno.

En el ejercicio número "10" habíamos colocado condición en el programa principal de tal forma que, si el número tecleado era menor que 155 y mayor que 27, se le sumaba 5 y si no se quedaba igual. Trataremos ahora de situar la condición en la subrutina:

Ejercicio "11":

Programa

```
   1 REM 00000000000000000000000
00000000000000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld hl,30000;ld e,(hl)
  40 REM call Suma
  45 REM ret;!fin principal
  50 REM Suma;ld a,155;cp e
  55 REM ret m,;!retorno condici
onal a rutina principal
  60 REM ld a,e;add a,5;ld (hl),
a
  70 REM ret;!retorno incondicio
nal a rutina principal
  80 REM finish
  90 RANDOMIZE USR 60000
 100 INPUT n
 105 IF n>255 OR n<28 THEN GO TO
 100
 110 POKE 30000,n
 120 RANDOMIZE USR 23760
 130 PRINT PEEK 30000
 140 GO TO 100
```

Comentarios

Atención: Observe la coma que sigue a **m** en **ret m,**

En todos los casos en que exista una condición de retorno, deberemos colocar la *coma* (,) indicada. Esta condición sintáctica es específica de nuestro ensamblador.

```
org 23760
23760 21 30 75      ld hl,30000
23763 5E            ld e,(hl)
23764 CD D8 5C      call Suma
23767 C9            ret
fin principal
Suma
23768 3E 9B         ld a,155
23770 BB            cp e
23771 F8            ret m,
retorno condicional a rutina p
23772 7B            ld a,e
23773 C6 05         add a,5
23775 77            ld (hl),a
23776 C9            ret
retorno incondicional a rutina
```

En la línea 55 se condiciona el retorno al programa principal al resultado de la comparación efectuada en la línea anterior, de forma que si tal comparación resulta negativa (*m*inus) se retorna al programa principal, en caso contrario sigue con la lectura de la misma hasta encontrar el *ret*orno *incondicional* de la línea 70.

Pseudo-operaciones (Primera parte)

Cualquier microprocesador, sea del tipo que sea, está diseñado de forma tal que puede realizar operaciones pertenecientes a un específico y determinado conjunto. Estos grupos de operaciones varían según los modelos de microprocesador y, por razones ya conocidas, aquí sólo se hará referencia a las pertenecientes al Z80.

Para que el microprocesador ejecute ésta o aquella operación deberemos darle la oportuna orden.

Cada una de estas órdenes reconocibles por el microprocesador y que le hace realizar una determinada operación recibe el nombre de **instrucción.**

El juego completo de instrucciones del Z80 lo puede encontrar al final del libro.

Estas instrucciones las capta el microprocesador por medio de los impulsos eléctricos que llegan –o no llegan– a las patillas destinadas a este fin.

Como ya se dijo en los primeros capítulos, nosotros representamos las instrucciones en forma de bytes o números binarios de 8 dígitos.

Para evitarnos la enojosa tarea de programar escribiendo instrucciones en forma de números binarios, aparece el lenguaje de asamblea y sus **nemónicos.**

Los nemónicos –o nemotécnicos– reemplazan las instrucciones por contracciones de vocablos ingleses cuyo significado recuerda la función que cumplen aquéllos.

Estos nemónicos sólo son interpretados por el **programa ensamblador**, el cual se encarga de transformarlos en los *bytes* equivalentes a las instrucciones que representan.

Dicho esto, podremos afirmar que el lenguaje de asamblea está pensado para hacer la vida más fácil al programador y, en este empeño, todos los programas ensambladores permiten el uso de ciertos nemotécnicos que no implican la realización de ningún tipo de operación por parte del microprocesador o, dicho de otra forma, no existe instrucción equivalente y cuyo objetivo es, o bien hacer operativo el propio programa ensamblador, o bien aumentar el grado de comodidad del usuario.

Este tipo de nemotécnicos son conocidos como **pseudo-op**, en razón –entiendo– de que parecen instrucciones destinadas a producir algún tipo de operación en el microprocesador, cuando –en realidad– sólo son obedecidas por el propio programa ensamblador.

Cada programa ensamblador nos permite usar sus propias **pseudo-op**, por esta razón deberemos conocer perfectamente todas aquéllas que nos ofrece el programa ensamblador que se vaya a utilizar y las posibilidades que cada uno brinda. En razón de esto echemos una ojeada a los rasgos más importantes del ensamblador Ultravioleta. El ensamblador Ultravioleta, como cualquier otro que se precie de tal, acepta y traduce a c.m. todas las instrucciones estándar del microprocesador Z80.

El campo de comentarios se puede utilizar en cualquier línea del listado, excepto en la inicial y final (**go** y **finish**). Como se habrá podido apreciar –y se ha dicho– todos los comentarios se inician con el símbolo ! y la longitud que aparecerá en los listados una vez traducidos a C.M. –ensamblado– será de 32 caracteres como máximo.

Con respecto a las etiquetas –*labels*– son válidos los comentarios ya hechos con las restricciones de no usar los caracteres) y +.

Otras características las encontrará en el manual del ensamblador Ultravioleta.

Pasemos ahora al objeto fundamental de este capítulo: Las *pseudo-operaciones.*

En la **Puesta en marcha** tratamos la pseudo-op **go** como una instrucción que interpreta el ensamblador para iniciar su función traductora del lenguaje de asamblea a lenguaje máquina. En otras palabras, el ensamblador no entra en acción hasta encontrar la pseudo-operación **go**. Eso es todo. Más explicaciones al respecto son innecesarias.

La pseudo-op **org** tiene una función muy definida y con una alternativa de uso sumamente práctica como ya veremos.

La *dirección origen* **org** asigna una dirección absoluta de memoria a partir de la cual se va a almacenar los bytes del código máquina equivalente al programa en lenguaje de asamblea. Esta pseudo-operación no se traduce a C.M., pero sí es interpretada por el ensamblador como una instrucción que posiciona la dirección origen a partir de la cual la memoria se va ocupando posición a posición.

En el capítulo MEMORIA, hablábamos de los diferentes sectores de memoria donde podemos ubicar nuestros programas en C.M. y, finalmente, bosquejamos el mejor proceso de trabajo y posterior almacenamiento en la zona alta de la memoria, para ello, la pseudo-op **org** nos provee de una opción de uso alternativa que estudiaremos detalladamente en el capítulo PSEUDO-OPERACIONES. SEGUNDA PARTE.

La última pseudo-op que veremos ahora es **finish** –en otros ensambladores puede encontrar **end**– y es de porte similar a **go**, con ésta lanzábamos el ensamblador y con aquélla lo paramos. Resumiendo, con **go** comenzamos a traducir a C.M. y con **finish** indicamos al ensamblador que su función traductora ha terminado.

Nada más de momento, ya volveremos sobre estos asuntos.

El primer gráfico

Por supuesto que nos quedan cosas por tratar pero, con lo aprendido hasta aquí, ya estamos en condiciones de realizar algún programa "movido" que nos haga ver que nuestro esfuerzo no es baldío.

En el libro *Los colores y los gráficos en el Spectrum* de esta misma colección podrá encontrar una detallada explicación sobre el *fichero de imágenes* y allí encontrará este programa:

```
10 FOR d=16384 TO 22527
20 POKE d,255
30 NEXT d
```

Al correrlo la pantalla se tornará, punto a punto, negra –o del color INK actual– siguiendo una cadencia establecida y a una velocidad impuesta por la velocidad de ejecución del BASIC.

A continuación vamos a estudiar un programa en lenguaje de asamblea cuya misión sea idéntica y que podrá cargar con el nombre "12" de la cinta de "Ejercicios".

```
  1 REM 00000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
 10 REM go
 20 REM org 23760
 30 REM ld hl,16384
 40 REM ld bc,6144
 50 REM ld a,0
 60 REM Pantalla;ld (hl),255
 70 REM inc hl
 80 REM call Retardo;ld d,150
 90 REM dec bc
100 REM cp b
110 REM jp nz,Pantalla
120 REM cp c
130 REM jp nz,Pantalla
140 REM ret;!fin principal
150 REM Retardo
160 REM Tiempo;dec d;cp d
170 REM ret z;;!fin subrrutina
180 REM jp Tiempo
190 REM finish
200 RANDOMIZE USR 60000
205 CLS
210 RANDOMIZE USR 23760
```

Antes de correr este programa, interpretaremos el listado y nos fijaremos en un par de detalles nuevos.

En la línea 30 cargamos el registro doble **hl** con el número 16384 que es la dirección de memoria donde comienza el *fichero de imágenes.*

En la línea 40 cargamos el registro doble **bc** con el número 6144 que es representativo del número de bytes que ocupa en memoria el fichero de imágenes. Ver MAPA DE MEMORIA.

Las líneas 50 y 60 son evidentes, no obstante observe que al cargar el contenido de la dirección dada por (**hl**) con 255, estamos llenando cada byte del fichero de imágenes con 11111111 –color INK– ya que en la línea siguiente se va aumentando en 1 sucesivamente la dirección dada en **hl**.

La línea 80 nos manda a una subrutina etiquetada bajo el nombre **Retardo** cuya misión es cubrir el bucle dado entre las líneas 150 y 180, tantas veces como indique el contenido del registro **d** que en nuestro ejemplo está cargado con 150. Vd. puede cambiarlo para comprobar la eficacia del bucle de retardo.

En el subprograma **Retardo** se produce el retorno sólo si el resultado de la comparación es 0, en caso contrario el **jp** incondicional de la línea 180 obliga a un salto a la etiqueta **Tiempo**.

Otra conclusión muy interesante que debemos sacar de este listado es cómo resolver una comparación entre un registro doble –del tipo **bc**– con el registro **a**.

Los registros de 16 bits son simplemente dos registros, el primero de los cuales guarda el byte más significativo (MSB) y el segundo el byte menos significativo (LSB), por tanto, entre ambos, podemos guardar un número comprendido entre 0 y 65535.

En función de lo anterior –y en este programa– hemos comparado el contenido del registro **b** –obviamente con el **a**–, si el resultado es NO CERO dirigimos la lectura a la etiqueta **Pantalla**, de otra forma sigue la lectura y comparamos **c** seguido de otro salto condicional similar al anterior.

Como apreciamos en la línea 90, el registro **bc** se decrementa –en uno– hasta alcanzar el valor 0, momento en el cual las dos comparaciones dan 0 y consiguientemente los saltos **jp nz** son superados llegando la lectura a la línea 140 concluyendo así la rutina en C.M.

Dicho esto, ensamblaremos nuestro programa (RUN, ENTER) y obtendremos:

```
org 23760
23760 21 00 40      ld hl,16384
23763 01 00 18      ld bc,6144
23766 3E 00         ld a,0
Pantalla
23768 36 FF         ld (hl),255
23770 23            inc hl
23771 CD DB 5C      call Retardo
23774 16 96         ld d,150
23776 0B            dec bc
23777 B8            cp b
23778 C2 E2 5C      jp nz,Pantall
23781 B9            cp c
23782 C2 E6 5C      jp nz,Pantall
23785 C9            ret
fin principal
Retardo
Tiempo
23786 15            dec d
23787 BA            cp d
23788 C8            ret z,
fin subrrutina
23789 C3 EA 5C      jp Tiempo
```

En estas condiciones –contenido del registro **d**, 150– la velocidad de impresión en pantalla es similar a la obtenida con el programa BASIC inicial pero, si cambiamos el contenido del registro **d** a 1, en el primer recorrido de la subrutina **Retardo** retornaremos al programa principal, con lo cual la velocidad de ejecución aumenta espectacularmente. Pruébelo. Compare las velocidades y vea si merece la pena seguir con el esfuerzo que exige aprender C.M.

Con la instrucción **ld (hl),255** de la línea 60, se cargan los bytes del fichero de imágenes como ya se ha dicho, con el modelo 11111111, pero ...¿qué pasaría si cambiamos este modelo por ϕ11111ϕ1 que equivale a 125 decimal? ¡Hágalo!

Cambie estos modelos por otros y saque sus conclusiones.

Dejemos este asunto y pasemos a Cervantes.

La historia de Don Quijote comienza por una E y esta letra será el objetivo del próximo capítulo.

La primera letra

En este capítulo vamos a aprender a escribir en la pantalla del televisor.

Algunos conceptos deben estar claros previamente:

En las páginas 183 y siguientes del manual de su Spectrum está la tabla denominada *El juego de caracteres*, si le echamos otra ojeada –y debemos hacerlo con tanta frecuencia que lleguemos a familiarizarnos con ella– veremos que la primera columna –*código*– está compuesta por una serie de números que, bajo ciertas condiciones que ahora estudiaremos, serán interpretadas para mostrar en pantalla el carácter que figura en la columna de la derecha –*carácter*–, siendo estos caracteres, propiamente los comprendidos entre los códigos 32 –espacio– y 164, correspondiente al último carácter definido por el usuario.

La columna cuarta corresponde a los nemónicos, a algunos de los cuales, si Vd. lo recuerda, ya les hemos puesto un puntito a su derecha...

En otro orden de cosas, pero relacionado con el tema que estamos tratando, debemos saber lo siguiente:

A. Con **call 5633** llamamos a una subrutina de la ROM que nos abre el CANAL 2, con lo cual se permite la impresión en la parte superior de la pantalla. Esta instrucción debe ir precedida de **ld a,2** ya que el canal abierto es, en definitiva, el que indica el contenido del registro **a**.

B. Con la instrucción **rst 16** se imprime en pantalla el carácter cuyo código esté cargado en el acumulador –**registro a**– en ese moemento.
Dicho esto pasemos a desarrollar un programa en lenguaje de asamblea que, finalmente, imprima en pantalla la letra **E**.

Ejercicio "13":

Programa

```
    1 REM 00000000000000000000000
00000000000000000000000000000000
   10 REM go
   20 REM org 23760
   30 REM ld a,2
   40 REM call 5633
   50 REM ld a,69
   60 REM rst 16
   70 REM ret
   80 REM finish
   90 RANDOMIZE USR 60000
   95 STOP
  100 RANDOMIZE USR 23760
```

```
org 23760
23760 3E 02        ld a,2
23762 CD 01 16     call 5633
23765 3E 45        ld a,69
23767 D7           rst 16
23768 C9           ret
```

Comentarios

Tanto **call** 5633 como **rst** 16 son subrutinas situadas en la ROM.

Como puede observar, en la línea 30 cargamos el acumulador con el número 2, lo cual permite a la instrucción **call 5633** abrir el canal de impresión correspondiente a la parte superior de la pantalla.

En la 50 cargamos el acumulador con el código del carácter **E** y, de esta forma, se permite que la subrutina **rst 16** imprima ese carácter en pantalla.

Con lo cual obtendremos una gloriosa E, tras superar el STOP de la línea 96.

Al final del capítulo anterior dijimos que la Historia del ingenioso hidalgo D. Quijote de La Mancha comienza con la letra E y nosotros sabemos ya cómo conseguir su impresión a través de un conjunto de instrucciones en lenguaje de asamblea.

Ahora escribamos el primer párrafo de nuestra mejor novela:

En un lugar de la Mancha
de cuyo nombre...

¿Preparado?

Bien, en primer lugar debemos notar que los códigos de las letras son distintos según éstas sean mayúsculas o minúsculas.

También debemos saber que los espacios en blanco se consiguen gracias al código del *SPACE* (32).

Con el código de ENTER (13) indicamos que acaba una línea de texto y comienza otra. Y, en general y como siempre, le recomiendo que conozca en profundidad el contenido de las páginas 183 y siguientes del Manual del Spectrum.

Teclee o carge el Ejercicio "14".

Programa

```
    1 REM 00000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
   10 REM go
   20 REM org 23760
   30 REM ld a,2
   40 REM call 5633
   50 REM ld a,69;rst 16;E
   60 REM ld a,110;rst 16;!n
   70 REM ld a,32;rst 16;!SPACE
   80 REM ld a,117;rst 16;!u
   90 REM ld a,110;rst 16;!n
  100 REM ld a,32;rst 16;!SPACE
  110 REM ld a,108;rst 16;!l
  120 REM ld a,117;rst 16;!u
  130 REM ld a,103;rst 16;!g
  140 REM ld a,97;rst 16;!a
  150 REM ld a,114;rst 16;!r
  160 REM ld a,32;rst 16;!SPACE
  170 REM ld a,100;rst 16;!d
  180 REM ld a,101;rst 16;!e
  190 REM ld a,32;rst 16;!SPACE
  200 REM ld a,108;rst 16;!l
  210 REM ld a,97;rst 16;!a
  220 REM ld a,32;rst 16;!SPACE
  230 REM ld a,77;rst 16;!M
  240 REM ld a,97;rst 16;!a
  250 REM ld a,110;rst 16;!n
  260 REM ld a,99;rst 16;!c
  270 REM ld a,104;rst 16;!h
```

Comentario

Observe cómo una vez abierto el canal 2 en 30 y 40, ya queda en esta situación y no necesitamos repetir estas instrucciones.

```
 280 REM ld a,97;rst 16;!a
 290 REM ld a,13;rst 16;!otra li
nea
 300 REM ld a,100;rst 16;!d
 310 REM ld a,101;rst 16;!e
 320 REM ld a,32;rst 16;!SPACE
 330 REM ld a,99;rst 16;!c
 340 REM ld a,117;rst 16;!u
 350 REM ld a,121;rst 16;!y
 360 REM ld a,111;rst 16;!o
 370 REM ld a,46;rst 16;!punto
 380 REM ld a,46;rst 16;!punto
 390 REM ld a,46;rst 16;!punto
 400 REM ret
 410 REM finish
 420 RANDOMIZE USR 60000
 430 RANDOMIZE USR 23760
```

Y finalmente tendremos:

```
org 23760
23760 3E 02          ld a,2
23762 CD 01 16       call 5633
23765 3E 45          ld a,69
23767 D7             rst 16
E
23768 3E 6E          ld a,110
23770 D7             rst 16
n
23771 3E 20          ld a,32
23773 D7             rst 16
SPACE
23774 3E 75          ld a,117
23776 D7             rst 16
u
23777 3E 6E          ld a,110
23779 D7             rst 16
n
23780 3E 20          ld a,32
23782 D7             rst 16
SPACE
23783 3E 6C          ld a,108
23785 D7             rst 16
l
23786 3E 75          ld a,117
23788 D7             rst 16
u
23789 3E 67          ld a,103
23791 D7             rst 16
g
23792 3E 61          ld a,97
23794 D7             rst 16
a
23795 3E 72          ld a,114
23797 D7             rst 16
r
23798 3E 20          ld a,32
23800 D7             rst 16
SPACE
23801 3E 64          ld a,100
23803 D7             rst 16
d
23804 3E 65          ld a,101
23806 D7             rst 16
e
23807 3E 20          ld a,32
23809 D7             rst 16
SPACE
23810 3E 6C          ld a,108
23812 D7             rst 16
l
23813 3E 61          ld a,97
23815 D7             rst 16
a
23816 3E 20          ld a,32
23818 D7             rst 16
SPACE
23819 3E 4D          ld a,77
23821 D7             rst 16
M
23822 3E 61          ld a,97
23824 D7             rst 16
a
23825 3E 6E          ld a,110
23827 D7             rst 16
n
23828 3E 63          ld a,99
23830 D7             rst 16
c
23831 3E 68          ld a,104
23833 D7             rst 16
h
23834 3E 61          ld a,97
23836 D7             rst 16
a
23837 3E 0D          ld a,13
23839 D7             rst 16
otra linea
23840 3E 64          ld a,100
23842 D7             rst 16
d
23843 3E 65          ld a,101
23845 D7             rst 16
e
23846 3E 20          ld a,32
23848 D7             rst 16
SPACE
23849 3E 63          ld a,99
23851 D7             rst 16
c
23852 3E 75          ld a,117
23854 D7             rst 16
u
23855 3E 79          ld a,121
23857 D7             rst 16
y
23858 3E 6F          ld a,111
23860 D7             rst 16
o
23861 3E 2E          ld a,46
23863 D7             rst 16
punto
23864 3E 2E          ld a,46
23866 D7             rst 16
punto
23867 3E 2E          ld a,46
23869 D7             rst 16
punto
23870 C9             ret
En un lugar de la Mancha
de cuyo...
```

Hasta aquí una introducción a la programación de textos.

En el próximo capítulo mezclaremos los conocimientos adquiridos hasta el momento, con algo de imaginación y así obtendremos...

Un poco de acción

La vista transmite a nuestro cerebro la sensación de movimiento continuo cuando interpreta una secuencia de imágenes consecutivas.

La forma de aprovechar este efecto óptico *utilizando los medios que nos brinda el computador*, está amplia y minuciosamente desarrollada en *Los colores y los gráficos en el Spectrum* y allí podemos encontrar este programa que hace aparecer y desaparecer una imagen con una velocidad controlada:

```
20 LET a=60
30 PRINT AT 0,0;"■";
40 FOR i=1 TO a: NEXT i
50 PRINT AT 0,0;" "
60 FOR j=1 TO a: NEXT j
70 GO TO 10
```

En la línea 30 imprimimos el carácter negro (■) y en la 50 lo hacemos "desaparecer" con un carácter predefinido "blanco".

Córralo y compruebe el efecto que produce.

Diseñemos un programa en lenguaje de asamblea que consiga lo mismo, pero desarrollándolo y comentándolo poco a poco.

En primer lugar, este pequeño listado nos imprimirá un negro (■) en el ángulo superior izquierdo de la pantalla.

Ejercicio "15":

```
    1 REM 000000000000000000000000
000000000000000000000000000000
   10 REM go
   20 REM org 23760
   30 REM ld a,2
   40 REM call 5633
   50 REM ld a,143
   60 REM rst 16
  190 REM ret
  200 REM finish
  210 RANDOMIZE USR 60000
  215 CLS
  220 RANDOMIZE USR 23760


org 23760
23760 3E 02          ld a,2
23762 CD 01 16       call 5633
23765 3E 8F          ld a,143
23767 D7             rst 16
23768 C9             ret

■
```

Al correrlo aparecerá en pantalla e inmóvil el carácter gráfico ■.

En estas condiciones si imprimiéramos otro carácter, aparecería en pantalla a la derecha del ■, pero si movemos el cursor a la izquierda (código 8, pág. 123 del manual) e imprimimos un carácter blanco sobre el negro anterior, el resultado debería ser el borrado inmediato del ■ dando la impresión, en definitiva, que nada se hubiera impuesto.

Añada las líneas 70 y 75 y pruebe

```
70 REM ld a, 8; rst 16
75 REN ld a, 128, rst 16
```

En la línea 70 movemos el cursor una posición a la izquierda y en la línea 75 imprimimos un carácter blanco, por tanto, encima del negro inicial. Conclusión, aparentemente no hay nada en la pantalla, aunque realmente hay un *blanco*.

En el siguiente listado añadimos un bucle para conseguir el efecto de parpadeo.

Ejercicio "16":

```
    1 REM 000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
   10 REM go
   20 REM org 23760
   30 REM ld a,2
   40 REM call 5633
   45 REM ld b,255
   47 REM Bucle
   50 REM ld a,143
   60 REM rst 16
   70 REM ld a,8;rst 16
   75 REM ld a,128;rst 16
   80 REM ld a,8;rst 16
   85 REM ld a,0;dec b;cp b
   86 REM jp nz,Bucle
  190 REM ret
  200 REM finish
  210 RANDOMIZE USR 60000
  214 STOP
  215 CLS
  220 RANDOMIZE USR 23760
```

```
org 23760
23760 3E 02          ld a,2
23762 CD 01 16       call 5633
23765 06 FF          ld b,255
Bucle
23767 3E 8F          ld a,143
23769 D7             rst 16
23770 3E 08          ld a,8
23772 D7             rst 16
23773 3E 80          ld a,128
23775 D7             rst 16
23776 3E 08          ld a,8
23778 D7             rst 16
23779 3E 00          ld a,0
23781 05             dec b
23782 B8             cp b
23783 C2 D7 5C       jp nz,Bucle
23786 C9             ret
```

Usted está en condiciones de interpretar todas las instrucciones, no obstante fíjese que además del bucle se ha añadido la línea 80 para mover el cursor atrás un carácter antes de iniciar el bucle nuevamente.

Córralo y verá parpadear, durante un breve período de tiempo, el cuadrado negro en la parte superior de la pantalla.

Evidentemente el parpadeo es excesivamente rápido y para verlo en cámara lenta vamos a introducir un bucle de Retardo.

En el ejemplo anterior pasábamos 255 veces por el **Bucle**, lo cual era aceptable en función de la velocidad del parpadeo, pero si introducimos una subrutina de **Retardo**, como hemos previsto, entonces será más adecuado reducir a 50 –**ld b,50**– el número de recorridos por el **Bucle**.

En función de estos comentarios cargue la cinta con el Ejercicio "17" o añada al "16" las líneas adecuadas.

Programa

```
    1 REM 00000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
   10 REM go
   20 REM org 23760
   30 REM ld a,2
   40 REM call 5633
   45 REM ld b,50
   47 REM Bucle
   50 REM ld a,143
   60 REM rst 16
   65 REM call Retardo;ld de,2000
0
   70 REM ld a,8;rst 16
   75 REM ld a,128;rst 16
   77 REM call Retardo;ld de,2000
0
   80 REM ld a,8;rst 16
   85 REM ld a,0;dec b;cp b
   86 REM jp nz,Bucle
  100 REM Retardo;ld a,0;dec de;c
p d;jp nz,Retardo;cp e;jp nz,Ret
ardo;ret
  190 REM ret
  200 REM finish
  210 RANDOMIZE USR 60000
  214 STOP
  215 CLS
  220 RANDOMIZE USR 23760
```

Comentarios

En las líneas 65 y 77 recurrimos a la subrutina **Retardo** e, inmediatamente después de volver de ella, cargamos el registro doble **de** con 20000 que es, en definitiva, el tiempo del retardo como se verá a continuación.
En la línea 100 establecemos el bucle **Retardo** en el cual condicionamos el retorno al programa principal al hecho de que los registros **d** y **e** sean 0, para lo cual decrementamos **de** y comparamos y obligamos a saltar nuevamente a **Retardo** gracias a las instrucciones **jp nz**.

```
org 23760
23760 3E 02        ld a,2
23762 CD 01 16     call 5633
23765 06 32        ld b,50
Bucle
23767 3E 8F        ld a,143
23769 D7           rst 16
23770 CD DA 5C     call Retardo
23773 11 20 4E     ld de,20000
23776 3E 08        ld a,8
23778 D7           rst 16
23779 3E 80        ld a,128
23781 D7           rst 16
23782 CD E6 5C     call Retardo
23785 11 20 4E     ld de,20000
23788 3E 08        ld a,8
23790 D7           rst 16
23791 3E 00        ld a,0
23793 05           dec b
23794 B8           cp b
23795 C2 D7 5C     jp nz,Bucle
Retardo
23798 3E 00        ld a,0
23800 1B           dec de
23801 BA           cp d
23802 C2 F6 5C     jp nz,Retardo
23805 BB           cp e
23806 C2 F6 5C     jp nz,Retardo
23809 C9           ret
23810 C9           ret
```

Ya tenemos un parpadeo con una cadencia similar al obtenido con el programa BASIC inicial.

Pruebe otras velocidades de parpadeo variando el contenido del par **de** en las líneas 65 y 77.

El siguiente paso será producir la sensación de desplazamiento del carácter negro, para lo cual lo haremos desaparecer, imprimiéndolo una posición más a la derecha. Ello implica, simplemente, quitar la línea 80 del listado anterior con lo que no hacemos retroceder el cursor al final de cada Bucle de parpadeo.

El nuevo listado sería

Ejercicio "18"

```
    1 REM 000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
   10 REM go
   20 REM org 23760
   30 REM ld a,2
   40 REM call 5633
   45 REM ld b,50
   47 REM Bucle
   50 REM ld a,143
   60 REM rst 16
   65 REM call Retardo;ld de,2000
0
   70 REM ld a,8;rst 16
   75 REM ld a,128;rst 16
   77 REM call Retardo;ld de,2000
0
   85 REM ld a,0;dec b;cp b
   86 REM jp nz,Bucle
  100 REM Retardo;ld a,0;dec de;c
p d;jp nz,Retardo;cp e;jp nz,Ret
ardo;ret
  190 REM ret
  200 REM finish
  210 RANDOMIZE USR 60000
  214 STOP
  215 CLS
  220 RANDOMIZE USR 23760
```

Ensámblelo y observe el avance del carácter gráfico.

Haga pruebas con otros caracteres, otros desplazamientos más largos y cuantas posibilidades vea Vd. de practicar.

Nuestro carácter gráfico ya se mueve por la pantalla a una velocidad controlada tanto en su parpadear como en su desplazamiento.

Pruebe a cambiar el contenido del par de registros **de** a la mitad y aún a valores inferiores.

Esta posibilidad de aumentar la velocidad hasta límites que hacen casi imperceptible la impresión, adquiere valor en toda su importancia cuando tratemos de conseguir varios movimientos simultáneos con vitalidad los cuales, en BASIC por ejemplo, parecerían imágenes de una película en cámara lenta.

Es claro que, una vez corrido el programa, nosotros no podemos actuar sobre lo que sucede en la pantalla. Y no podemos porque ya es momento de pasar al capítulo siguiente para conocer ciertas técnicas relativas al teclado.

A través del teclado

El teclado es el único dispositivo que nos permite introducir información en el ordenador justo en el momento que lo consideramos conveniente. Nos sirve tanto para teclear los listados de programas, como para actuar sobre éstos una vez corridos. En consecuencia, se hace imprescindible conocer ciertas técnicas que nos permitan dominar su capacidad operativa.

Como ya veremos con más detalle posteriormente, cada tecla es chequeada 50 veces por segundo para saber si ha sido –o no– apretada, en caso afirmativo el código del carácter correspondiente pasa al acumulador. Toda esta tarea ha sido encomendada a una subrrutina de la ROM, la cual mantiene informado constantemente al microprocesador sobre la situación en que se encuentra el teclado.

Unos ejemplos explicados con todo detalle abrirán nuestra capacidad lógica al respecto.

Supongamos que queremos desarrollar una rutina en C.M. que mantenga el teclado inhabilitado excepto si apretamos la letra **a** y que, en ese momento, se imprima en pantalla ese carácter.

En el programa que desarrollamos a continuación recurrimos a la subrutina **call 703**, situada en la ROM, cuya labor es la descrita unos párrafos más arriba.

En las primeras líneas abrimos el canal 2 para imprimir en la parte superior de la pantalla y en el resto se chequea el teclado una y otra vez, hasta que la tecla **a** (código 97) sea apretada. En ese momento el programa procede a imprimir el contenido del registro **a** que, según lo explicado, debe ser el código del último carácter tecleado.

Ejercicio "19"

```
    1 REM 000000000000000000000000
00000000000000000000000000000
   10 REM go
   20 REM org 23760
   21 REM ld a,2
   22 REM call 5633
   40 REM Test;call 703
   43 REM cp 97;jp nz,Test
   75 REM rst 16
   80 REM ret
   90 REM finish
  100 RANDOMIZE USR 60000
  105 CLS
  110 RANDOMIZE USR 23760

org 23760
23760 3E 02          ld a,2
23762 CD 01 16       call 5633
Test
23765 CD BF 02       call 703
23768 FE 61          cp 97
23770 C2 D5 5C       jp nz,Test
23773 D7             rst 16
23774 C9             ret
```

Observe cómo en la línea 40 obligamos a chequear el código de la tecla que se prieta –si es que se aprieta alguna– y en la línea 43 pasamos a comparar directamente el código de la letra deseada con el contenido –claro está– del acumulador, sin que previamente hayamos introducido ninguna instrucción específica de carga del registro **a**, y esto es debido a que el acumulador se carga automáticamente con el código del carácter de la tecla apretada en el constante rastreo que hace la instrucción **call 703** en el teclado.

Si Vd. corre este programa, la pantalla se quedará en blanco y, apriete la tecla que apriete, así continuará a menos que sea la letra **a**. Pruebe y haga sus propios ensayos.

Apoyándonos en el procedimiento anterior, desarrollaremos un programa que haga parpadear un carácter negro en la pantalla hasta el momento en que apretemos la tecla *BREAK* (Space-código 32).

Ejercicio "20"

```
   1 REM 00000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
0000000
  10 REM go
  20 REM org 23760
  50 REM ld a,2
  60 REM call 5633
  70 REM Test;call 703;cp 32;jp
nz,Prog
 100 REM ret;!Fin principal
 140 REM Prog
 150 REM ld a,143;rst 16
 155 REM call Retardo
 170 REM ld a,8;rst 16
 180 REM ld a,128;rst 16
 190 REM call Retardo
 200 REM ld a,8;rst 16
 210 REM jp Test
 250 REM Retardo;ld a,0;dec bc;c
p b;jp nz,Retardo;cp c;jp nz,Ret
ardo;ld bc,60000;ret
 300 REM finish
 310 RANDOMIZE USR 60000
 314 STOP
 315 CLS
 320 RANDOMIZE USR 23760
```

```
org 23760
23760 3E 02          ld a,2
23762 CD 01 16       call 5633
Test
23765 CD BF 02       call 703
23768 FE 20          cp 32
23770 C2 DA 5C       jp nz,Prog
23773 C9             ret
Fin principal
Prog
23774 3E 8F          ld a,143
23776 D7             rst 16
23777 CD E1 5C       call Retardo
23780 3E 08          ld a,8
23782 D7             rst 16
23783 3E 80          ld a,128
23785 D7             rst 16
23786 CD EA 5C       call Retardo
23789 3E 08          ld a,8
23791 D7             rst 16
23792 C3 D5 5C       jp Test
Retardo
23795 3E 00          ld a,0
23797 0B             dec bc
23798 B8             cp b
23799 C2 F3 5C       jp nz,Retardo
23802 B9             cp c
```

```
23803 C2 F3 5C      jp nz,Retardo
23806 01 60 EA      ld bc,60000
23809 C9            ret
```

Después de las **pseudo-op** imprescindibles, abrimos el canal de impresión ya conocido para, a continuación, crear una etiqueta **Test** con la que, en caso de que la tecla **SPACE** sea apretada, se dé por terminada la ejecución del programa (Línea 100). De otra forma se salta a la etiqueta **Prog**, a partir de la cual se genera un listado de parpadeo con retardo del tipo ya conocido.

Teniendo ya una forma de *salir* del programa a voluntad, hagamos que nuestro negro cuadradito se desplace a la derecha cuando la tecla L –por ejemplo– sea apretada.

Empecemos por fijar el carácter en el ángulo superior izquierdo:

Ejercicio "21"

```
   1 REM 00000000000000000000000
00000000000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld a,2;call 5633
 100 REM Test;call 703;cp 32;jp
nz,Prog
 150 REM ret;!Final
 200 REM Prog
 250 REM ld a,143;rst 16
 260 REM ld a,8;rst 16
 500 REM jp Test
1010 REM finish
2000 RANDOMIZE USR 60000
2005 CLS
2010 RANDOMIZE USR 23760
```

```
org 23760
23760 3E 02         ld a,2
23762 CD 01 16      call 5633
Test
23765 CD BF 02      call 703
23768 FE 20         cp 32
23770 C2 DE 5C      jp nz,Prog
23773 C9            ret
Final
Prog
23774 3E 8F         ld a,143
23776 D7            rst 16
23777 3E 08         ld a,8
23779 D7            rst 16
23780 C3 D5 5C      jp Test
```

Hemos creado un bucle de impresión y marcha atrás del cursor que sólo se romperá en el caso de apretar la tecla **SPACE**, con lo cual conseguimos la impresión fija.

Ampliemos el listado anterior, para conseguir una sucesión de caracteres negros, dando la impresión de trazar una gruesa línea a medida que apretamos la tecla **1**.

Ejercicio "22"

```
   1 REM 00000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld a,2;call 5633
 100 REM Test;call 703;cp 32;jp
z,Stop;cp 108;jp z,Raya;jp Prog
 145 REM Stop
 150 REM ret;!Final
```

```
 200 REM Prog
 250 REM ld a,143;rst 16
 260 REM ld a,8;rst 16
 500 REM jp Test
 750 REM Raya;ld a,143;rst 16
 800 REM jp Test
1010 REM finish
2000 RANDOMIZE USR 60000
2005 STOP
2006 CLS
2010 RANDOMIZE USR 23760
```

```
org 23760
23760 3E 02          ld a,2
23762 CD 01 16       call 5633
Test
23765 CD BF 02       call 703
23768 FE 20          cp 32
23770 CA DA 5C       jp z,Stop
23773 FE 6C          cp 108
23775 CA DF 5C       jp z,Raya
23778 C3 E2 5C       jp Prog
Stop
23781 C9             ret
Final
Prog
23782 3E 8F          ld a,143
23784 D7             rst 16
23785 3E 08          ld a,8
23787 D7             rst 16
23788 C3 EC 5C       jp Test
Raya
23791 3E 8F          ld a,143
23793 D7             rst 16
23794 C3 D5 5C       jp Test
```

La línea 100 es la clave de este listado y de los sucesivos, ya que en ella chequeamos el teclado de forma que si la tecla apretada es **SPACE** obligamos a saltar a la etiqueta **Stop**, *saliendo* del programa, si, por el contrario apretamos la tecla *L*, el programa sigue en la etiqueta **Raya** con lo cual imprimimos caracteres uno detras de otro. Finalmente, y en la misma línea, si ninguno de los saltos anteriores ha sido aceptado, la lectura sigue en la etiqueta **Prog**, cuyo efecto es el ya observado en el ejercicio previo. Al correr este programa verá que la velocidad es excesiva y para ralentizarlo, introducimos un bucle de retardo:

Ejercicio "23"

```
   1 REM 000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld a,2;call 5633
 100 REM Test;call 703;cp 32;jp
z,Stop;cp 108;jp z,Raya;jp Prog
 145 REM Stop
 150 REM ret;!Final
 200 REM Prog
 250 REM ld a,143;rst 16
 260 REM ld a,8;rst 16
 500 REM jp Test
 750 REM Raya;ld a,143;rst 16
 769 REM ld de,5000
 770 REM Retardo;ld a,0;dec de;c
p d;jp nz,Retardo;cp e;jp nz,Ret
ardo
 800 REM jp Test
1010 REM finish
2000 RANDOMIZE USR 60000
2005 CLS
2010 RANDOMIZE USR 23760
```

```
org 23760
23760 3E 02          ld a,2
23762 CD 01 16       call 5633
Test
23765 CD BF 02       call 703
23768 FE 20          cp 32
23770 CA DA 5C       jp z,Stop
23773 FE 6C          cp 108
23775 CA DF 5C       jp z,Raya
23778 C3 E2 5C       jp Prog
Stop
23781 C9             ret
Final
Prog
23782 3E 8F          ld a,143
23784 D7             rst 16
23785 3E 08          ld a,8
23787 D7             rst 16
23788 C3 EC 5C       jp Test
Raya
23791 3E 8F          ld a,143
23793 D7             rst 16
23794 11 88 13       ld de,5000
Retardo
23797 3E 00          ld a,0
23799 1B             dec de
23800 BA             cp d
23801 C2 F5 5C       jp nz,Retardo
23804 BB             cp e
23805 C2 F5 5C       jp nz,Retardo
23808 C3 D5 5C       jp Test
```

Para aumentar o disminuir la velocidad varíe el contenido del par de registro **de** en la línea 769. Córralo y compruebe diferentes velocidades.

Y por último vamos a ir borrando caracteres negros a medida que se producen sus impresiones en pantalla, de forma tal que obtengamos la sensación de desplazamiento, según se apriete –o no– la tecla L para ir hacia la derecha y la A para la izquierda.

Ejercicio "24"

```
   1 REM 000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld a,2;call 5633
 100 REM Test;call 703;cp 32;jp
z,Stop;cp 108;jp z,Derecha;cp 97
;jp z,Izquierda;jp Prog
 145 REM Stop
 150 REM ret;!Final
 200 REM Prog
 250 REM ld a,143;rst 16
 260 REM ld a,8;rst 16
 500 REM jp Test
 750 REM Derecha;ld hl,23688;ld
b,(hl);ld a,2;cp b;jp z,Test
 760 REM ld a,128;rst 16;ld a,14
3;rst 16;ld a,8;rst 16
 769 REM Tiempo;ld de,5000
 770 REM Retardo;ld a,0;dec de;c
p d;jp nz,Retardo;cp e;jp nz,Ret
ardo
 800 REM jp Test
 900 REM Izquierda;ld hl,23688;l
d b,(hl);ld a,33;cp b;jp z,Test
 910 REM ld a,8;rst 16;ld a,143;
rst 16;ld a,128;rst 16;ld a,8;rs
t 16;ld a,8;rst 16
 950 REM jp Tiempo
1010 REM finish
2000 RANDOMIZE USR 60000
2006 CLS
2010 RANDOMIZE USR 23760
```

```
org 23760
23760 3E 02        ld a,2
23762 CD 01 16     call 5633
Test
23765 CD BF 02     call 703
23768 FE 20        cp 32
23770 CA DA 5C     jp z,Stop
23773 FE 6C        cp 108
23775 CA DF 5C     jp z,Derecha
23778 FE 61        cp 97
23780 CA E4 5C     jp z,Izquierd
23783 C3 E7 5C     jp Prog
Stop
23786 C9           ret
Final
Prog
23787 3E 8F        ld a,143
23789 D7           rst 16
23790 3E 08        ld a,8
23792 D7           rst 16
23793 C3 F1 5C     jp Test
Derecha
23796 21 88 5C     ld hl,23688
23799 46           ld b,(hl)
23800 3E 02        ld a,2
23802 B8           cp b
23803 CA FB 5C     jp z,Test
23806 3E 80        ld a,128
23808 D7           rst 16
23809 3E 8F        ld a,143
23811 D7           rst 16
23812 3E 08        ld a,8
23814 D7           rst 16
Tiempo
23815 11 88 13     ld de,5000
Retardo
23818 3E 00        ld a,0
23820 1B           dec de
23821 BA           cp d
23822 C2 0E 5D     jp nz,Retardo
23825 BB           cp e
23826 C2 12 5D     jp nz,Retardo
23829 C3 15 5D     jp Test
Izquierda
23832 21 88 5C     ld hl,23688
23835 46           ld b,(hl)
23836 3E 21        ld a,33
23838 B8           cp b
23839 CA 1F 5D     jp z,Test
23842 3E 08        ld a,8
23844 D7           rst 16
23845 3E 8F        ld a,143
23847 D7           rst 16
23848 3E 80        ld a,128
23850 D7           rst 16
23851 3E 08        ld a,8
23853 D7           rst 16
23854 3E 08        ld a,8
23856 D7           rst 16
23857 C3 31 5D     jp Tiempo
```

Este programa es básicamente igual al anterior, con la diferencia de las líneas 750, 760, 900, 910 y 950. En la línea 100, se ha cambiado la etiqueta denominada Raya por **Derecha**, que será el sentido del desplazamiento del carácter y, por similar razón, se ha introducido la etiqueta **Izquierda**, a la cual se saltará en el caso de que apretemos la tecla A.

También se ha añadido, en la línea 769, la etiqueta **Tiempo** para evitar generar un bucle de retardo nuevo después de cada paso por la etiqueta **Izquierda**, como se podrá ver.

Dicho esto, pasemos a hacer unos comentarios orientativos sobre la función que cumple cada una de las líneas adicionales ya mencionadas.

Línea 750:

En el manual del Spectrum, dentro del epígrafe *Las Variables del Sistema*, página 176, tenemos la dirección 23688, la cual contiene la posición horizontal del cursor en el momento actual. Para comprobar esto, comience por limpiar su pantalla.

Como sabemos, existen 32 posiciones posibles horizontales de carácter –32 columnas– que van de la 0 a la 31, lo cual implica que, horizontalmente, el cursor que fija la siguiente impresión esperada debe guardar el dato de esa posición en algún lugar.

Ese lugar es la dirección de memoria 23688 ya mencionada.

Si tecleamos

PRINT AT 0, 0 ; PEEK 23688

nos aparecerá en el ángulo superior izquierdo de pantalla (posición 0,0) en 33 que es el contenido de la citada posición de memoria cuando el cursor está lo más a la izquierda posible.

Un PRINT AT 0, 31; PEEK 23688

nos dará un 2 en la esquina superior derecha que es el contenido de esa dirección cuando el cursor está lo más a la derecha posible.

De lo que sigue, la primera conclusión que debe sacar es la conveniencia de conocer las *variables del sistema* y su función para, de esta forma, aumentar el conjunto de sus recursos a la hora de programar.

Volviendo a las instrucciones de la línea objeto de estos comentarios observará que cargamos el registro **a**, con el valor 2 y lo comparamos con el contenido de la dirección 23688, si el resultado es 0 devolvemos el control a la etiqueta **Test**, en otro caso se permite la secuencia normal de lectura del programa. Esto quiere decir en términos prácticos que, si el carácter negro que estamos desplazando a la derecha llega al límite derecho de la pantalla se parará. Hemos puesto una condición de borde.

En la **línea 900** hemos puesto otra condición de borde, igual a la anterior, pero a la izquierda.

La **línea 760**, borra el carácter anterior superponiendo un blanco, imprimiendo un negro a continuación y dando marcha atrás al cursor para restablecer la situación en el caso de que la lectura pasara a la etiqueta **Prog**. Mientras se mantenga la tecla L apretada se borra e imprime, de forma que se obtiene la sensación de desplazamiento a la derecha.

La línea **910** actúa de forma similar para el desplazamiento a la izquierda.

En la línea **950** se ha introducido un salto a la etiqueta **Tiempo** para evitarnos teclear un bucle de retardo exactamente igual al que se inicia en la línea 769 y que acaba con un salto incondicional a la etiqueta **Test**.

Para acabar este capítulo listamos a continuación un programa que permite mover el **negro** a derecha, izquierda y hacia abajo según se aprieten las teclas L, A ó Z... ¿Se atrevería a añadir las líneas que faltan para conseguir que suba cuando se apriete la tecla P, imponiendo la condición de borde correspondiente?

Como podrá apreciar, al final de este listado se ha modificado el carácter gráfico "a" para conseguir el efecto de un pequeño tranvía en movimiento.

Una vez tecleado o cargado el ejercicio "25", comience, como es lógico por un GOTO 3000.

```
   1 REM 000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
  10 REM go
  20 REM org 23760
  30 REM ld a,2;call 5633
 100 REM Test;call 703;cp 32;jp
z,Stop;cp 108;jp z,Derecha;cp 97
;jp z,Izquierda;cp 122;jp z,Abaj
o;jp Prog
 145 REM Stop
 150 REM ret;!Final
 200 REM Prog
 250 REM ld a,144;rst 16
 260 REM ld a,8;rst 16
 500 REM jp Test
 750 REM Derecha;ld hl,23688;ld
b,(hl);ld a,2;cp b;jp z,Test
 760 REM ld a,128;rst 16;ld a,14
4;rst 16;ld a,8;rst 16
 769 REM Tiempo;ld de,5000
 770 REM Retardo;ld a,0;dec de;c
p d;jp nz,Retardo;cp e;jp nz,Ret
ardo
 800 REM jp Test
 900 REM Izquierda;ld hl,23688;l
d b,(hl);ld a,33;cp b;jp z,Test
 910 REM ld a,8;rst 16;ld a,144;
rst 16;ld a,128;rst 16;ld a,8;rs
t 16;ld a,8;rst 16
 920 REM jp Tiempo
1000 REM Abajo;ld hl,23689;ld b,
(hl);ld a,4;cp b;jp z,Test
1010 REM ld a,128;rst 16;dec (hl
);ld a,8;rst 16;ld a,144;rst 16;
ld a,8;rst 16
1030 REM jp Tiempo
1999 REM finish
2000 RANDOMIZE USR 60000
2005 CLS
2010 RANDOMIZE USR 23760
2020 STOP
3000 POKE USR "a"+0,BIN 11111111
3100 POKE USR "a"+1,BIN 00000000
3200 POKE USR "a"+2,BIN 00000000
3300 POKE USR "a"+3,BIN 11111111
3400 POKE USR "a"+4,BIN 11111111
3500 POKE USR "a"+5,BIN 00000000
3600 GO TO 2000
```

```
org 23760
23760 3E 02          ld a,2
23762 CD 01 16       call 5633
Test
23765 CD BF 02       call 703
23768 FE 20          cp 32
23770 CA DA 5C       jp z,Stop
23773 FE 6C          cp 108
23775 CA DF 5C       jp z,Derecha
23778 FE 61          cp 97
23780 CA E4 5C       jp z,Izquierd
23783 FE 7A          cp 122
23785 CA E9 5C       jp z,Abajo
23788 C3 EC 5C       jp Prog
Stop
23791 C9             ret
Final
Prog
23792 3E 90          ld a,144
23794 D7             rst 16
```

```
23795 3E 08        ld a,8
23797 D7           rst 16
23798 C3 F6 5C     jp Test
Derecha
23801 21 88 5C     ld hl,23688
23804 46           ld b,(hl)
23805 3E 02        ld a,2
23807 B8           cp b
23808 CA 00 5D     jp z,Test
23811 3E 80        ld a,128
23813 D7           rst 16
23814 3E 90        ld a,144
23816 D7           rst 16
23817 3E 08        ld a,8
23819 D7           rst 16
Tiempo
23820 11 88 13     ld de,5000
Retardo
23823 3E 00        ld a,0
23825 1B           dec de
23826 BA           cp d
23827 C2 13 5D     jp nz,Retardo
23830 BB           cp e
23831 C2 17 5D     jp nz,Retardo
23834 C3 1A 5D     jp Test
Izquierda
23837 21 88 5C     ld hl,23688
23840 46           ld b,(hl)
23841 3E 21        ld a,33
23843 B8           cp b
23844 CA 24 5D     jp z,Test
23847 3E 08        ld a,8
23849 D7           rst 16
23850 3E 90        ld a,144
23852 D7           rst 16
23853 3E 80        ld a,128
23855 D7           rst 16
23856 3E 08        ld a,8
23858 D7           rst 16
23859 3E 08        ld a,8
23861 D7           rst 16
23862 C3 0C 5D     jp Tiempo
Abajo
23865 21 89 5C     ld hl,23689
23868 46           ld b,(hl)
23869 3E 04        ld a,4
23871 B8           cp b
23872 CA D5 5C     jp z,Test
23875 3E 80        ld a,128
23877 D7           rst 16
23878 35           dec (hl)
23879 3E 08        ld a,8
23881 D7           rst 16
23882 3E 90        ld a,144
23884 D7           rst 16
23885 3E 08        ld a,8
23887 D7           rst 16
23888 C3 0C 5D     jp Tiempo
```

Observe la ampliación de la línea 100. Trate de ver el *funcionamiento* de las líneas 1000, 1010 y 1020.

Pseudo-operaciones (Segunda parte)

Al finalizar la *primera parte* del epígrafe dedicado a este tema unas páginas más atrás, quedamos en continuar aquí.

Y comenzaremos tratando un tema de evidente interés.

Si ya sabemos cómo introducir un programa en lenguaje de asamblea –código máquina–, cómo modificarlo cuando nos interesa y cómo utilizarlo; muy probablemente nos preguntaremos... ¿en qué área de la memoria –y cómo– se guarda el programa una vez traducido a código máquina, de forma que nos deje libre la zona de memoria dedicada al programa en BASIC?

La respuesta a esta pregunta pasa por comprender el funcionamiento de la pseudo-op denominada **org** en una variante sumamente práctica, cuya sintaxis corresponde al esquema:

org *dirección origen actual (dirección origen deseada).*

Para hacer claro el concepto, recurramos al Ejercicio "24" el cual tiene su dirección origen en 23760 y supongamos que queremos reubicarlo a partir de la dirección 60120.

Para recorrer todo el proceso, cargue el *ensamblador* y a continuación el Ejercicio "24", con este programa en memoria borre las instrucciones 2010 y 2006, puesto que ya no van a tener objeto y cambie la línea **20 REM org 23760** por **20 REM org 23760 (60120)**, dejando un espacio entre ambos números y sin olvidar el paréntesis según figura.

Ensamble el programa "24" con estas modificaciones realizadas y observe cómo han cambiado las direcciones de memoria desde la 60120 (origen) hasta la 60220.

Coja una cinta virgen o, en todo caso, distinta de su original de *Ejercicios* y disponga todo para efectuar la grabación del programa "24" y teclee, a continuación:

SAVE "nombre" *CODE* 23760, 100

donde *SAVE* y *CODE* son sentencias del BASIC localizables en el teclado, *"nombre"* es el título asignado al programa, *23760* es la dirección de origen del programa en código máquina actual y *100* es el número de bytes que ocupa el programa, lo cual es deducible fácilmente del listado ensamblado del Ejercicio "24".

Efectúe la grabación.

Desconecte el Spectrum.

Cargue la grabación anterior con LOAD "nombre" CODE 60120, 100.

Aparecerá en pantalla.

Bytes: nombre.

Ya tenemos el código máquina a partir de la dirección 6ϕ12ϕ, para probarlo introduzca y corra este pequeño programa en BASIC.

```
10 FOR x=60120 TO 60220
20 PRINT x,PEEK x
30 NEXT x
40 CLS
50 RANDOMIZE USR 60120
60 STOP
```

Con la primera instrucción sacamos un listado de las microinstrucciones y datos, en decimal, del programa en código máquina. A continuación llamamos, desde el BASIC, a la subrutina en C.M. situada en la dirección 60120.

Y, finalmente, sólo resta decir que la pseudo-operación **org** puede ser usada en tantas ocasiones como queramos hacer saltar el programa *ensamblador* a nuevas direcciones origen.

Antes de pasar a la siguiente pseudo-op, haremos un ejercicio que comenzaremos con la carga del *ensamblador* y a continuación el Ejercicio "24". Una vez hecho esto cambiaremos **org** para reubiarlo, p.e., en 61000 y borraremos la línea 2010.

Ahora tecleemos las líneas comprendidas entre la 35 y 97 según muestra esta parte inicial del listado del "24" ya modificado:

```
   1 REM 00000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
  10 REM go
  20 REM org 23760 (61000)
  30 REM ld a,2;call 5633
  35 REM nop
  40 REM nop
  45 REM nop
  50 REM nop
  55 REM nop
  60 REM nop
  65 REM nop
  70 REM nop
  75 REM nop
  80 REM nop
  85 REM nop
  90 REM nop
  95 REM nop
  97 REM nop
 100 REM Test;call 703;cp 32;jp
z,Stop;cp 108;jp z,Derecha;cp 97
;jp z,Izquierda;jp Prog
 145 REM Stop
 150 REM ret;!Final
 200 REM Prog
 250 REM ld a,143;rst 16
 260 REM ld a,8;rst 16
 500 REM jp Test
 750 REM Derecha;ld hl,23688;ld
b,(hl);ld a,2;cp b;jp z,Test
 760 REM ld a,128;rst 16;ld a,14
3;rst 16;ld a,8;rst 16
 769 REM Tiempo;ld de,5000
 770 REM Retardo;ld a,0;dec de;c
p d;jp nz,Retardo;cp e;jp nz,Ret
ardo
 800 REM jp Test
 900 REM Izquierda;ld hl,23688;l
d b,(hl);ld a,33;cp b;jp z,Test
 910 REM ld a,8;rst 16;ld a,143;
rst 16;ld a,128;rst 16;ld a,8;rs
t 16;ld a,8;rst 16
 950 REM jp Tiempo
1010 REM finish
2000 RANDOMIZE USR 60000
2006 CLS
```

nop significa NO OPERA y es un nuevo nemónico que implica dejar la posición de memoria que le corresponde sin nada, de tal forma que, al ensamblar el programa con estas instrucciones, nos aparecerán esas direcciones con contenido 0, según muestra el listado.

Bueno ¿y qué? Se dirá Vd.

Sin nervios. Despacito.

Un programa en C.M., como ya sabemos, es una secuencia de direcciones de memoria rellenas cada una de ellas con microinstrucciones y datos, esto quiere decir que, si no dejáramos –gracias a **nop**– algunas posiciones de memoria vacías, nos sería imposible introducir nuevas instrucciones o datos, a menos que comencemos todo el proceso de ensamblado y reubicación.

En el caso del ejercicio "24" que ya reubicamos a partir de la posición 60120, si hubiéramos tenido la precaución de introducir estos **nop**, las modificaciones que vamos a introducir hubieran sido directas. Vamos a simular el proceso con el programa modificado de acuerdo con las últimas indicaciones y que ensamblado nos dará:

```
org 23760 (61000)
61000 3E 02        ld a,2
61002 CD 01 16     call 5633
61005 00           nop
61006 00           nop
61007 00           nop
61008 00           nop
61009 00           nop
61010 00           nop
61011 00           nop
61012 00           nop
61013 00           nop
61014 00           nop
61015 00           nop
61016 00           nop
61017 00           nop
61018 00           nop
Test
61019 CD BF 02     call 703
61022 FE 20        cp 32
61024 CA 60 EE     jp z,Stop
61027 FE 6C        cp 108
61029 CA 65 EE     jp z,Derecha
61032 FE 61        cp 97
61034 CA 6A EE     jp z,Izquierd
61037 C3 6D EE     jp Prog
Stop
61040 C9           ret
Final
Prog
61041 3E 8F        ld a,143
61043 D7           rst 16
61044 3E 08        ld a,8
61046 D7           rst 16
61047 C3 77 EE     jp Test
Derecha
61050 21 88 5C     ld hl,23688
61053 46           ld b,(hl)
61054 3E 02        ld a,2
61056 B8           cp b
61057 CA 81 EE     jp z,Test
61060 3E 80        ld a,128
61062 D7           rst 16
61063 3E 8F        ld a,143
61065 D7           rst 16
61066 3E 08        ld a,8
61068 D7           rst 16
Tiempo
61069 11 88 13     ld de,5000
Retardo
61072 3E 00        ld a,0
61074 1B           dec de
61075 BA           cp d
61076 C2 94 EE     jp nz,Retardo
61079 BB           cp e
```

```
61080 C2 98 EE      jp nz,Retardo
61083 C3 9B EE      jp Test
Izquierda
61086 21 88 5C      ld hl,23688
61089 46            ld b,(hl)
61090 3E 21         ld a,33
61092 B8            cp b
61093 CA 5B EE      jp z,Test
61096 3E 08         ld a,8
61098 D7            rst 16
61099 3E 8F         ld a,143
61101 D7            rst 16
61102 3E 80         ld a,128
61104 D7            rst 16
61105 3E 08         ld a,8
61107 D7            rst 16
61108 3E 08         ld a,8
61110 D7            rst 16
61111 C3 8D EE      jp Tiempo
```

En función de este listado sabemos que el programa ocupa 114 bytes (61111 – 61000 + 3 bytes finales).

Grabemos el programa en una cinta aparte con un

SAVE "cursor" CODE 23760, 114

y a continuación carguemos el Spectrum con un

LOAD "cursor" CODE 61000, 114

Cuando este proceso finalice tendremos el programa *cursor* ubicado a partir de la línea 61000 y con las direcciones de memoria 61005 a 61018 disponibles, es decir, 14 posiciones.

Si en estas condiciones tecleamos y corremos el siguiente programa en BASIC.

```
  10 FOR x=60005 TO 61013
  20 READ n
  30 POKE x,n
  40 DATA 62,143,215,62,143,215,
62,143,215
  50 NEXT x
  60 PRINT "tres negros comenzar
 en..."
  70 INPUT "Quieres probar?(s/n)
";r$
  80 IF r$="s" THEN RANDOMIZE US
R 61000
  90 PRINT "Adios"
```

rellenamos las 9 direcciones indicadas con los octetos necesarios para imprimir tres caracteres negros.

Es conveniente aclarar, antes de seguir, que **nop** es un nemónico y no una pseudo-operación.

En estos momentos tenemos en el computador un programa en BASIC y una rutina en código máquina.

¿Cómo haremos para grabar ambos de forma que al cargar el programa "CURSOR" en BASIC, todo quede dispuesto para su uso? Para distinguir los programas en BASIC de aquellos otros en código máquina, escribiremos los nombres de los primeros en mayúsculas y los segundos en minúsculas.

En primer lugar, al programa BASIC le añadiremos la línea

5 LOAD" "CODE 61000, 114

El siguiente paso será grabar el BASIC con

SAVE "CURSOR" LINE 5

Con lo cual el programa se autoejecuta a partir de la línea 5.

Cuando el mensaje 0 OK, 0:1 aparezca, paramos el grabador y tecleamos SAVE "cursor" CODE 61000, 114 y grabamos de nuevo.

Finalizada esta operación, desconecte el Spectrum y proceda a la carga, como es usual, con un LOAD "CURSOR".

Primero aparecerá en pantalla:

Program: CURSOR

y después

Bytes: cursor

Proceso finalizado.

Pasamos a otra pseudo-op.

El efecto de **equ** es simple: iguala una dirección a una etiqueta, con la gran ventaja de que podemos recurrir a ella desde cualquier otra subrutina extraña al programa actual, sin más que referirnos al nombre de la etiqueta.

La sintaxis de esta seudo-op es

equ *dirección etiqueta.*

En otras palabras, si en una subrutina en lenguaje de asamblea nos referimos a una etiqueta que no esté definida dentro de la propia subrutina, inevitablemente nos encontraremos con un error que impedirá el proceso de ensamblado, para evitar esto, tendríamos que dirigir el *salto* al número que represente la dirección de memoria a la cual nos queremos dirigir.

Es claro que resulta más cómodo y práctico memorizar el nombre de una etiqueta como p.e, **Parpadeo** que la dirección donde esté situada y que podría ser 25639. Si a esto le añade que un programa largo se confecciona a *trozos* que posteriormente, y una vez unidos, darán cuerpo al mismo, se encontrará en la necesidad de recurrir a muchas direcciones que están situadas fuera del *trozo* que actualmente esté desarrollando, con lo cual aumenta la conveniencia de utilizar etiquetas.

Cargue el ensamblador.

Para el fin que nos proponemos, recurriremos a un ejemplo sencillo pero suficientemente demostrativo de cómo utilizar **equ**.

Desarrollamos a continuación un programa, en dos trozos, para desplazar el cursor –representado por un negro– a la derecha.

Ejercicio "26"

```
   1 REM 00000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
0
  10 REM go
  20 REM org 23760
  21 REM equ 25000 Derecha
  30 REM ld a,2;call 5633
 100 REM Test;call 703;cp 32;jp
z,Stop;cp 108;call z,Derecha;jp
Prog
 145 REM Stop
 150 REM ret;!Final
 200 REM Prog
 250 REM ld a,143;rst 16
 260 REM ld a,8;rst 16
 270 REM jp Test
 510 REM org 25000
 550 REM equ 30000 Tiempo
 750 REM Derecha;ld hl,23688;ld
b,(hl);ld a,2;cp b;ret z
 760 REM ld a,128;rst 16;ld a,14
3;rst 16;ld a,8;rst 16
 769 REM call Tiempo;ld de,5000
 800 REM ret
4000 REM finish
4010 RANDOMIZE USR 60000
4015 CLS
4020 RANDOMIZE USR 23760
```

```
org 23760
equ 25000 Derecha
23760 3E 02        ld a,2
23762 CD 01 16     call 5633
Test
23765 CD BF 02     call 703
23768 FE 20        cp 32
23770 CA DA 5C     jp z,Stop
23773 FE 6C        cp 108
23775 CC DF 5C     call z,Derech
23778 C3 E2 5C     jp Prog
Stop
23781 C9           ret
Final
Prog
23782 3E 8F        ld a,143
23784 D7           rst 16
23785 3E 08        ld a,8
23787 D7           rst 16
23788 C3 EC 5C     jp Test
org 25000
equ 30000 Tiempo
Derecha
25000 21 88 5C     ld hl,23688
25003 46           ld b,(hl)
25004 3E 02        ld a,2
25006 B8           cp b
25007 00           ret z
25008 3E 80        ld a,128
25010 D7           rst 16
25011 3E 8F        ld a,143
25013 D7           rst 16
25014 3E 08        ld a,8
25016 D7           rst 16
25017 CD 30 75     call Tiempo
25020 11 88 13     ld de,5000
25023 C9           ret
```

Tres cosas deben llamarle la atención.

Primero, si se ejecuta este programa, fallará.

En segundo lugar, en las líneas 20 y 510 hemos colocado dos diferentes *direcciones origen* en 23760 y 25000, lo que equivale a dos rutinas diferentes sin más comunicación entre ellas que la establecida gracias a la pseudo-op *equ*.

En tercer lugar, en la línea 21 hemos igualado la dirección 25000 –lugar donde comienza la segunda subrutina– con la etiqueta **Derecha** para evitar que el ensamblador nos dé un error al llegar a la línea 100 donde, entre otras cosas, se le pide que vaya a la subrutina **Derecha** y que está situada fuera de la rutina actual.

Observe cómo en la línea 769 se llama a una subrutina –*Tiempo*– que no está definida en el programa más que a través de la *equ* de la línea 550. Esto implica que *debe* haber un tercer *trozo* de programa a partir de la línea 30000, que produzca el deseado efecto de retardo. Añádalo Vd.

Esta tercera subrutina debe responder a la siguiente idea y, desde luego, situada a partir de la dirección 30000. Fíjese en los modelos precedentes ya utilizados.

```
Retardo
ld de, 20000
ld a, 0
cp d
jp nz, Retardo
cp e
jp nz, retardo
ret
```

Otra aplicación práctica de *equ* se deriva de la posibilidad de utilizarlo como una constante, situada en una posición de memoria, y a la cual nos referimos por su nombre.

El siguiente programa nos dará idea de cómo aplicar esta alternativa:

```
    1 REM 00000000000000000000000
000000000
   10 REM go
   20 REM org 23760
   50 REM equ 30000 Caracter
   60 REM ld a,2;call 5633
   70 REM ld a,(Caracter);rst 16
  160 REM ret
  170 REM finish
  180 RANDOMIZE USR 60000
  185 STOP
  190 RANDOMIZE USR 23760

org 23760
equ 30000 Caracter
23760 3E 02        ld a,2
23762 CD 01 16     call 5633
23765 3A 30 75     ld a,(Caracte
23768 D7           rst 16
23769 C9           ret
```

Dada la configuración del anterior listado, antes de correrlo, introduzca en la dirección 30000 el código del carácter que desee. Si tecleamos **POKE 30000,97**, nos aparecería una **a** en el ángulo superior derecho, una vez ejecutado el programa. Esto es todo respecto a **equ**. A continuación estudiaremos dos pseudo-op de reserva de memoria, con funciones distintas.

Con **defb** se rellena una posición de memoria, con el número que vaya detrás de ella. Este número debe estar comprendido entre 0 y 255.

La utilidad de esta instrucción se hace notar cuando, a lo largo de un programa, nos interese tener una serie de valores en unas posiciones de memoria correlativas, cuya primera dirección podemos definir mediante una etiqueta.

En base a estas explicaciones veamos prácticamente la función asignada a **defb**.

Ejercicio "27".

```
    1 REM 00000000000000000000000
000000000
   10 REM go
   20 REM org 23760
   50 REM Prueba;defb 22;defb 26
  defb 50;defb 143
  100 REM ret
  110 REM finish
  120 RANDOMIZE USR 60000
  125 CLS
  150 FOR x=23760 TO 23763
  160 PRINT x,PEEK x
  170 NEXT x
```

```
org 23760
Prueba
defb 22
defb 26
defb 50
defb 143
23764 C9            ret
```

Las tres últimas líneas del programa en BASIC nos dan la comprobación del contenido de las direcciones de memoria reservadas y rellenadas con *defb* en la línea 50.

```
23760          22
23761          26
23762          50
23763          143
```

La última pseudo-op que trataremos es **defs**, también de reserva de memoria.

Con ella imprimiremos mensajes de una forma cómoda en lenguaje de asamblea, de tal manera que el ensamblador coloca el código de cada carácter de los que siguen a **defs**, en posiciones de memoria consecutivas y a partir de la actual. Un ejemplo similar al anterior clarificará la función que cubre esta pseudo-op.

Ejercicio "28".

```
    1 REM 00000000000000000000000
000000000000000000000
   10 REM go
   20 REM org 23760
   50 REM Prueba;defs Cadena cara
cteres
  160 REM ret
  170 REM finish
  180 RANDOMIZE USR 60000
  185 CLS
  190 RANDOMIZE USR 23760
  200 FOR x=23760 TO 23776: PRINT
  x,CHR$ PEEK x: NEXT x
```

```
org 23760
Prueba
defs Cadena caracteres
23777 C9            ret
```

Gracias a la línea 200, obtenemos el contenido de las posiciones de memoria afectadas por *defs*:

```
23760          C
23761          a
23762          d
23763          e
23764          n
23765          a
23766
23767          c
23768          a
23769          r
23770          a
23771          c
23772          t
23773          e
23774          r
23775          e
23776          s
```

En los capítulos siguientes se verán aplicaciones prácticas de estas pseudo-operaciones, para las cuales necesitaremos comprender unos cuantos conceptos más.

Un aviso. A medida que nos acercamos al final –y ya estamos cerca– el control de este libro pasa, poco a poco, a sus manos. Quiere esto decir que yo sólo puedo mostrarle conceptos concretos y algunas alterantivas de uso, pero el conjunto de posibilidades que ya tiene Vd. son tan grandes, que sólo su iniciativa y su imaginación pueden ponerle límite.

La pila

Pila en su acepción de amontonar, de apilar, y es en este sentido en el que utilizamos esta expresión.

La *pila* es una zona de la memoria usada, tanto por el microprocesador como por el programador, para el almacenamiento eventual de información.

El microprocesador *guarda* direcciones de memoria. El programador puede *guardar* los valores que actualmente contengan los registros.

Al comienzo del libro, al hablar sobre los registros, se comentó el **sp** (stack pointer) o **puntero de pila.**

El registro **sp** tiene por misión *señalar* cuál es la siguiente posición libre dentro de la pila, de forma tal que, si algún dato nuevo entra en ella, se coloca exactamente en el lugar de la pila que indica el registro **sp**, desplazándose éste para señalar la siguiente dirección y si por el contrario la información sale de la pila el puntero se desplaza una posición de memoria para señalar la posición anterior.

En realidad el puntero salta dos posiciones adelante o hacia atrás, señalando siempre la primera posición vacía, puesto que todos los datos en la pila están conformados por dos bytes.

De aquí podemos deducir que la información contenida en la pila se maneja de acuerdo con el criterio de que el *último dato que entra en la pila es el primero que sale.*

Los nemónicos que cumplen la función de poner información dentro de la pila o sacarla son

push y **pop**

que reflejan la idea física de estas operaciones:

EMPUJA y TIRA

Por otra parte, el registro denominado **pc** —contador de programa— contiene constantemente la próxima dirección de memoria a *leer* por el microprocesador.

Así, cuando en la secuencia normal de lectura de un programa aparece una posición de memoria con una instrucción **call** a una dirección en la cual comienza una subrutina, el **pc** transfiere su contenido al **sp** y el **pc** se rellena con la dirección de la primera posición de memoria de la subrutina a ejecutar. En este momento, el programa sigue leyendo las direcciones de memoria de la subrutina hasta que encuentra la instrucción **ret**. Ahora el contenido correspondiente al **sp** pasa al **pc**, con lo cual la lectura sigue a partir de la dirección de memoria en que fue abandonado el programa principal.

Bien, ésta es una historia que no nos interesaría más que a nivel de información, si no fuera porque, como veremos a continuación, Vd. puede alterar la posición del **sp** con lo cual dislocaría

el lugar de retorno de la subrutina a la cual ha dirigido al microprocesador, se pueden haber introducido (*push*) a sacado (*pop*) datos en la pila y en estos casos habría que restablecer la situación antes de alcanzar el *ret*.

Sobre cómo debe ser manejada la pila por el programador trataremos en lo que sigue.

La sintaxis de los nemónicos que nos afectan es la siguiente:

push *rr* **pop** *rr*

rr representa cualquier par de registros.

Con **push** *rr* introducimos el contenido del par de registros *rr* en la *pila*.

Con **pop** *rr* sacamos el contenido correspondiente de la *pila* al par de registros *rr*.

Con **ld sp, NN** creamos una pila a partir de la dirección de memoria **NN**.

Con **ex (sp), hl** intercambiamos el último dato introducido en la pila con el contenido del par de registros **hl**.

Unos ejercicios pondrán fin a este capítulo.

Carge el Ejercicio "28", borre la línea 200 y añada las líneas suficientes como para igualarlo al siguiente listado del Ejercicio "29".

```
    1 REM 00000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
0
   10 REM go
   20 REM org 23760
   50 REM Prueba;defs Cadena cara
cteres
   60 REM push af
   70 REM push bc
   80 REM push hl
   90 REM ld a,2;call 5633
  100 REM ld b,17;!Numero caracte
res
  110 REM ld hl,Prueba
  120 REM Loop;ld a,(hl);rst 16
  130 REM inc hl
  140 REM djnz Loop
  150 REM pop hl;pop bc;pop af
  160 REM ret
  170 REM finish
  180 RANDOMIZE USR 60000
  185 CLS
  190 RANDOMIZE USR 23760
```

```
org 23760
Prueba
defs Cadena caracteres
23777 F5            push af
23778 C5            push bc
23779 E5            push hl
23780 3E 02         ld a,2
23782 CD 01 16      call 5633
23785 06 11         ld b,17
Numero caracteres
23787 21 EB 5C      ld hl,Prueba
Loop
23790 7E            ld a,(hl)
23791 D7            rst 16
23792 23            inc hl
```

```
23793 10 FE        djnz Loop
23795 E1           pop hl
23796 C1           pop bc
23797 F1           pop af
23798 C9           ret
```

que finalmente imprimirá en pantalla:

```
Cadena caracteres
```

¡Ajá! ¿Conque hay cosas que no entiende? Pues, ya tiene una buena razón para pasar al siguiente capítulo.

No obstante, comentaremos algunos temas que sí estamos en condiciones de comprender.

Observe que entre las líneas 90 y 140 hemos utilizado los registros **a, b** y **hl**, lo cual necesariamente implica que habremos alterado el contenido que hubieran podido tener los registros **a, f, bc** y **hl**, entonces, recurriendo a la pila, introducimos sus contenidos en la misma con las líneas 60, 70 y 80 y, una vez ejecutadas las instrucciones comprendidas entre 90 y 140 –que ya estudiaremos– se restablece el contenido de los pares de registros afectados, gracias a la línea 150.

De esta forma podríamos haber seguido programando a continuación de la línea 150, como si nada hubiera ocurrido a los registros **a, f, bc** y **hl**.

Un truco derivado de todo lo anterior, proviene del hecho de no poder transferir datos directamente de un par de registros a otro par, pero sí podemos pasar primero de un registro de 16 bits a la pila y, a continuación, de ésta al par de registro destinatarios.

Ejercicio "30"

```
   1 REM 000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
000
  10 REM go
  20 REM org 23760
  30 REM ld bc,62349
  40 REM push bc
  70 REM pop de
  80 REM ld hl,32001;ld (hl),d;d
ec hl;ld (hl),e
  90 REM ret
 100 REM finish
 110 RANDOMIZE USR 60000
 120 CLS
 130 RANDOMIZE USR 23760
 140 PRINT PEEK 32000+256*PEEK 3
2001


org 23760
23760 01 8D F3     ld bc,62349
23763 C5           push bc
23764 D1           pop de
23765 21 01 7D     ld hl,32001
23768 72           ld (hl),d
23769 2B           dec hl
23770 73           ld (hl),e
23771 C9           ret



62349
```

El anterior ejercicio confirma la transferencia del contenido del registro *bc* al *de*, a través de la instrucción en BASIC de la línea 140 y de la REM de la línea 80.

Bucles

A lo largo de muchos de los ejecricios anteriores nos hemos referido a bucles. Y los hemos utilizado.

El juego de instrucciones del Z80 nos provee de una forma cómoda, en comparación con la aplicada hasta el momento por nosotros, de realizar los bucles.

En el Ejercicio "29" se ha ejecutado un bucle de acuerdo con la técnica que se va a explicar a continuación y refiriéndonos a ese ejercicio.

En la línea 100 se ha cargado el registro *b* con una cantidad que es el número de veces que el bucle debe ser recorrido.

En la línea 140 aparece un nuevo nemónico cuya sintaxis es la siguiente:

djnz *etiqueta*

Y que viene a significar para el microprocesador: decrementa (en 1) el contenido del registro **b** y salta a la *etiqueta* si el contenido del registro **b** es *no cero* (**nz**).

La instrucción *djnz* sólo puede ser usada en combinación con el registro **b**.

En este orden de cosas, nada impide bucles dentro de bucles –o anidado de bucles– tantas veces como sea necesario.

Las señalizaciones. El registro f

Registro **f**	7	6	5	4	3	2	1	0	posición del bit
	s	z		h		p/v	n	c	nombre del bit

Sobre este registro ya iniciamos la introducción en el capítulo **SALTANDO**, de forma que a estas alturas ya nos resultan familiares las *banderas* (flags) **z** y **s** y el camino para servirnos de sus funciones.

Un vistazo general al resto de los bits nos pondrá al corriente de las posibilidades que nos ofrece el registro **f**.

El *bit* 0 contiene la *bandera* **c** y nos es de gran utilidad para sumar, restar y comparar.

Como es lógico, este bit sólo puede tener dos valores 0 y 1. Para dejar **c** a 1 utilizamos el nemónico **scf** y para dejarlo a 0 nos pueden valer **and, or** y **xor**. Es importante poner a 0 esta bandera antes de utilizarlo. El nemónico **ccf** nos deja este bit con el valor contrario al actual.

Para ver en qué forma actúa **c** en las operaciones aritméticas, basta recordar que cuando nosotros sumanos dos números, p.e., 8 y 7, decimos: *ocho más siete igual a quince y me llevo una.* Esa unidad que nos *llevamos* –y que los chavales recuerdan estirando un dedo– el microprocesador, en binario, la guarda en la bandera **c**, de tal forma que si, al sumar dos números binarios, el resultado es menor que 256 –es decir es un número comprendido entre 0 y 255 decimal (00000000 y 11111111)– no nos llevamos nada y **c** contiene un ϕ, en otro caso contiene un 1.

Esto será de aplicación un poco más adelante.

En las comparaciones, la bandera *c* se torna 0 cuando el resultado de restar del acumulador el contenido de cualquier otro registro es mayor o igual a 0, en otro caso es 1.

También interviene en las rotaciones...

El *bit 1* contiene la *bandera* **n** y nos indica –con un 1– si la última operación realizada por el microprocesador ha sido una resta.

El *bit 2* contiene la *bandera* **v** y la **p**. La **v** indica si se ha sobrepasado el campo numérico de los bytes signados. La **p** es el indicador de paridad y contiene 0 si la cantidad de *unos* en un byte es impar y 1 si es par.

Para condicionar saltos, llamadas a rutinas y retornos utilizaremos **po** para impar y **pe** para par.

El *bit 3* y el *5* no actúan.

El *bit 4* contiene la *bandera* **h** y funciona exactamente igual que la **c** pero sobre los cuatro primeros bits de los dos octetos implicados.

Los *bits 6* y *7* y su cometido ya han sido comentados.

En el resto del curso utilizaremos algunas de estas banderas pero, en cualquier caso, la intención de este capítulo es orientarle con las funciones que cubre el registro **f**, de forma que en el transcurso de sus propias investigaciones sobre modos de programar en lenguaje máquina, sepa a qué atenerse con respecto a las posibilidades de las diferentes *banderas*.

Un poco de matemáticas. 8 bits

En el transcurso de todas las páginas pasadas, prácticamente no hemos necesitado de las matemáticas para poder ir manejándonos, pero las circunstancias exigen un esfuerzo adicional.

Y si bien es verdad que el que más sabe es el que más puede aprender, no es menos cierto que el que más aprende es el que así lo desea y con esta filosofía vamos a introducirnos en un nuevo y muy sencillo sistema para sumar, restar, ... y los conocimientos accesorios que le son inherentes.

Una aclaración previa será de gran utilidad.

En los capítulos dedicados a la impresión en pantalla y a la introducción de información *a través del teclado*, se han utilizado ciertas subrutinas que nos han permitido cargar el acumulador con el código ASCII del carácter cuya tecla se haya apretado o hayamos decidido sacar por pantalla. La tabla ASCII la encontrará al final del libro.

Pero es el caso que, cuando ese carácter es representativo de un dígito (del 0 al 9), en el acumulador está contenido el código ASCII y en *ningún caso* el valor numérico correspondiente. Esto quiere decir que si introducimos datos numéricos desde el teclado a un programa, tendremos que transformar el código representativo del carácter del dígito a su valor numérico.

Si por el contrario necesitamos sacar por pantalla un dígito, resultado de cualquier proceso aritmético, tendríamos que pasar del valor numérico del dígito a su código ASCII.

En términos de programación esta transformación es sencilla, basta con restar 48 del código del carácter para obtener su valor numérico o sumar 48 al valor numérico para lograr el código ASCII. Todo lo anterior está referido a dígitos, es decir, números comprendidos entre 0 y 9.

El siguiente listado sigue este proceso en detalle. Trate de entenderlo antes de pasar a los comentarios.

Ejercicio "31":

```
    1 REM 000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
000000000000000
   10 REM go
   20 REM org 23760
   50 REM ld a,2;call 5633
   60 REM Test;call 703;cp 32;jp
z,Stop;cp 56;jp p,Test;cp 48;jp
m,Test;jp Prog
   80 REM Stop
   90 REM ret;!Final
  100 REM Prog
  120 REM sub 48;!Pasa a numerico
  125 REM ld hl,30000;ld (hl),a
  130 REM add a,2;!Suma dos
  140 REM add a,48;!Pasa a codigo
```

```
150 REM rst 16
160 REM ld a,13;rst 16
170 REM ret;!Final alternativo
190 REM finish
210 RANDOMIZE USR 60000
215 CLS
220 RANDOMIZE USR 23760
225 PRINT PEEK 30000
```

```
org 23760
23760 3E 02          ld a,2
23762 CD 01 16       call 5633
Test
23765 CD BF 02       call 703
23768 FE 20          cp 32
23770 CA DA 5C       jp z,Stop
23773 FE 38          cp 56
23775 F2 DF 5C       jp p,Test
23778 FE 30          cp 48
23780 FA E4 5C       jp m,Test
23783 C3 E7 5C       jp Prog
Stop
23786 C9             ret
Final
Prog
23787 D6 30          sub 48
Pasa a numerico
23789 21 30 75       ld hl,30000
23792 77             ld (hl),a

23793 C6 02          add a,2
Suma dos
23795 C6 30          add a,48
Pasa a codigo
23797 D7             rst 16
23798 3E 0D          ld a,13
23800 D7             rst 16
23801 C9             ret
Final alternativo
```

En la línea 60 imponemos condiciones al teclado.

En la línea 120 transformamos el código del carácter en su valor numérico. Todo sucede en el acumulador.

La línea 125 junto con la 225 sirven para comprobar la transformación.

En la línea 130 sumamos 2 al contenido del acumulador.

En la línea 140 transformamos el número en su código ASCII correspondiente. Y en la 160 iniciamos otra línea para la siguiente impresión, que se producirá en la 225.

Otra aclaración previa también de gran utilidad.

Tabla para sumar números binarios:

0 + 0 = 0 y *no me llevo nada*

1 + 0 = 1 y *no me llevo nada*

0 + 1 = 1 y *no me llevo nada*

1 + 1 = 0 y *me llevo una*

En otro orden de cosas, y aproximándonos un poco más al objeto de este capítulo, vamos a ver qué sucede con los números negativos.

Todos los números que hemos manejado hasta el momento han sido positivos y para nada se necesitó el signo "–".

También es conveniente recordar que con 8 bits –en un byte– podemos representar números binarios comprendidos entre 0 0 0 0 0 0 0 0 y 1 1 1 1 1 1 1 1 ó, lo que es igual, entre 0 y 255 en decimal.

Justo en el momento que tengamos que **signar** un byte, deberemos renunciar al bit situado más a la izquierda para dedicarlo a representar el signo, de tal suerte que el campo numérico que abarcará el byte en estas condiciones quedará comprendido entre –128 y +127.

En otras palabras, el bit situado más a la izquierda, como cualquier otro, sólo puede ser 0 ó 1. Si es ϕ, nos indica que el número es positivo, si es 1, negativo.

Es claro, que la pérdida de este bit en favor del signo, nos ha dejado con una representación numérica limitada a los siete bits restantes situados más a la derecha.

En función de lo anterior el campo de los números positivos admitido por un byte está comprendido entre

0 0 0 0 0 0 0 0 B	y	0 1 1 1 1 1 1 1 B	equivalente a
0	y	127	decimal.

los cuales explícitamente son:

1 = 00000001	2 = 00000010	3 = 00000011	4 = 00000100	5 = 00000101
6 = 00000110	7 = 00000111	8 = 00001000	9 = 00001001	10 = 00001010
11 = 00001011	12 = 00001100	13 = 00001101	14 = 00001110	15 = 00001111
16 = 00010000	17 = 00010001	18 = 00010010	19 = 00010011	20 = 00010100
21 = 00010101	22 = 00010110	23 = 00010111	24 = 00011000	25 = 00011001
26 = 00011010	27 = 00011011	28 = 00011100	29 = 00011101	30 = 00011110
31 = 00011111	32 = 00100000	33 = 00100001	34 = 00100010	35 = 00100011
36 = 00100100	37 = 00100101	38 = 00100110	39 = 00100111	40 = 00101000
41 = 00101001	42 = 00101010	43 = 00101011	44 = 00101100	45 = 00101101
46 = 00101110	47 = 00101111	48 = 00110000	49 = 00110001	50 = 00110010
51 = 00110011	52 = 00110100	53 = 00110101	54 = 00110110	55 = 00110111
56 = 00111000	57 = 00111001	58 = 00111010	59 = 00111011	60 = 00111100
61 = 00111101	62 = 00111110	63 = 00111111	64 = 01000000	65 = 01000001
66 = 01000010	67 = 01000011	68 = 01000100	69 = 01000101	70 = 01000110
71 = 01000111	72 = 01001000	73 = 01001001	74 = 01001010	75 = 01001011
76 = 01001100	77 = 01001101	78 = 01001110	79 = 01001111	80 = 01010000
81 = 01010001	82 = 01010010	83 = 01010011	84 = 01010100	85 = 01010101
86 = 01010110	87 = 01010111	88 = 01011000	89 = 01011001	90 = 01011010
91 = 01011011	92 = 01011100	93 = 01011101	94 = 01011110	95 = 01011111
96 = 01100000	97 = 01100001	98 = 01100010	99 = 01100011	100 = 01100100
101 = 01100101	102 = 01100110	103 = 01100111	104 = 01101000	105 = 01101001
106 = 01101010	107 = 01101011	108 = 01101100	109 = 01101101	110 = 01101110
111 = 01101111	112 = 01110000	113 = 01110001	114 = 01110010	115 = 01110011
116 = 01110100	117 = 01110101	118 = 01110110	119 = 01110111	120 = 01111000
121 = 01111001	122 = 01111010	123 = 01111011	124 = 01111100	125 = 01111101
126 = 01111110	127 = 01111111			

Los negativos correspondientes son aquéllos que sumados a su positivo nos dan como resultado 0.

Por todo lo expuesto parece lógico que el negativo de 00011010B (26 decimal) sea 10011010B (154 decimal), si esto fuera así, tendríamos el siguiente contrasentido:

0 0 0 1 1 0 1 0 B	equivale a	+26 decimal
1 0 0 1 1 0 1 0 B	suponemos equivale a	+ –26 decimal
1 0 1 1 0 1 0 0 B →	equivalente →	–52 decimal
	incorrecto	

lo cual implica que la representación del número negativo no ha sido correcta ya que el resultado de sumar a un número su negativo no es cero.

El proceso para obtener el negativo de un binario pasa por las siguientes etapas

1.º número positivo

0 0 0 1 1 0 1 0 B <> + 26 decimal

2.º Invertimos los bits, es decir, los 0 los pasamos a *1*, y los 1 a 0.

1 1 1 0 0 1 0 1 B

3.º Al número anterior le sumamos 0 0 0 0 0 0 0 1 B.

```
  1 1 1 0 0 1 0 1 B
+ 0 0 0 0 0 0 0 1 B
-------------------
  1 1 1 0 0 1 1 0 B   <>   -26
```

Hagamos la prueba de sumar este número y su negativo.

```
  0 0 0 1 1 0 1 0 B   <>   +26 decimal
+ 1 1 1 0 0 1 1 0 B   <>   -26 decimal
-----------------------------------
  0 0 0 0 0 0 0 0 B   <>   0
```

La unidad que nos llevamos al finalizar la suma se da por perdida sólo en este caso.

Este proceso es el que sigue el microprocesador para determinar el número negativo del valor contenido en el acumulador –siempre que esté comprendido entre 0 y 127–, al aplicar el nemónico **neg**.

Vd. puede determinar este valor de forma inmediata, restando de 128 el positivo dado. Siguiendo con el número de nuestro ejemplo:

128 – 26 = 102 0 1 1 0 0 1 1 0 B

y cambiando el 0 del bit más significativo por el 1 representativo del signo –:

1 1 1 0 0 1 1 0 B

De aquí podemos determinar que el campo de los números negativos admitido por un byte está comprendido entre:

1 1 1 1 1 1 1 1 B y 1 0 0 0 0 0 0 0 B
–1 –128

y son:

						128 =	10000000	129 =	10000001	130 =	10000010
131	10000011	132 =	10000100	133 =	10000101	134 =	10000110	135 =	10000111		
136 =	10001000	137 =	10001001	138 =	10001010	139 =	10001011	140 =	10001100		
141 =	10001101	142 =	10001110	143 =	10001111	144 =	10010000	145 =	10010001		
146 =	10010010	147 =	10010011	148 =	10010100	149 =	10010101	150 =	10010110		
151 =	10010111	152 =	10011000	153 =	10011001	154 =	10011010	155 =	10011011		
156 =	10011100	157 =	10011101	158 =	10011110	159 =	10011111	160 =	10100000		
161 =	10100001	162 =	10100010	163 =	10100011	164 =	10100100	165 =	10100101		
166 =	10100110	167 =	10100111	168 =	10101000	169 =	10101001	170 =	10101010		

171 = 10101011	172 = 10101100	173 = 10101101	174 = 10101110	175 = 10101111
176 = 10110000	177 = 10110001	178 = 10110010	179 = 10110011	180 = 10110100
181 = 10110101	182 = 10110110	183 = 10110111	184 = 10111000	185 = 10111001
186 = 10111010	187 = 10111011	188 = 10111100	189 = 10111101	190 = 10111110
191 = 10111111	192 = 11000000	193 = 11000001	194 = 11000010	195 = 11000011
196 = 11000100	197 = 11000101	198 = 11000110	199 = 11000111	200 = 11001000
201 = 11001001	202 = 11001010	203 = 11001011	204 = 11001100	205 = 11001101
206 = 11001110	207 = 11001111	208 = 11010000	209 = 11010001	210 = 11010010
211 = 11010011	212 = 11010100	213 = 11010101	214 = 11010110	215 = 11010111
216 = 11011000	217 = 11011001	218 = 11011010	219 = 11011011	220 = 11011100
221 = 11011101	222 = 11011110	223 = 11011111	224 = 11100000	225 = 11100001
226 = 11100010	227 = 11100011	228 = 11100100	229 = 11100101	230 = 11100110
231 = 11100111	232 = 11101000	233 = 11101001	234 = 11101010	235 = 11101011
236 = 11101100	237 = 11101101	238 = 11101110	239 = 11101111	240 = 11110000
241 = 11110001	242 = 11110010	243 = 11110011	244 = 11110100	245 = 11110101
246 = 11110110	247 = 11110111	248 = 11111000	249 = 11111001	250 = 11111010
251 = 11111011	252 = 11111100	253 = 11111101	254 = 11111110	255 = 11111111

Con la única consideración de que el orden decimal cambia de esta forma: el 255 equivale al –1, el 254 al –2, y así sucesivamente.

Las mismas formas vistas para determinar el negativo de un positivo binario comprendido entre 0 y 127, pueden ser utilizadas para determinar el positivo de un negativo dado, supongamos que se desea determinar el positivo de 1 1 1 1 1 0 1 1 B (–5 decimal).

Para ello invertimos los bits y sumamos 1

```
  00000100B
+ 00000001B
-----------
  00000101B <>   +5
```

ó también

128 – (–5) = 128 + 5 = 133 <> 10000101 B

y cambiando el bit de signo

00000101 B <> +5

El proceso de inversión de bit y posterior adición de 1 es conocido como **complemento a dos** y es exactamente la función que desarrolla la instrucción **neg**.

Si todo lo expuesto sobre los bytes signados y no signados quedó claro, deberá ser evidente que es elección del programador si un byte conforma un número de 8 dígitos binarios o de 7 más un bit de signo.

Por tanto, según decidamos, tenemos dos opciones para los mismos números:

SIN SIGNAR

```
  00011010B   equivalente a    26 decimal
+ 11010110B   equivalente a   214 decimal
-----------                  ----------
  11110000B   equivalente a   240 decimal
```

SIGNADO

```
  00011010B   equivalente a   +26 decimal
+ 11010110B   equivalente a   –42 decimal
-----------                  ----------
  11110000B   equivalente a   –16 decimal
```

En ambos casos el resultado de la suma realizada por el microprocesador es el mismo número, pero lo puede interpretar como

1 1 1 1 0 0 0 0 B <> n.º de 8 dígitos binarios

ó bien como

signo → 1 1 1 1 0 0 0 0 B <> n.º de 7 dígitos binarios con signo.

Unos párrafos más atrás, cuando comentábamos el proceso para determinar el complemento a dos, decíamos que se da por perdida la unidad que nos llevábamos al finalizar la suma.

También vimos en la Tabla para sumar números binarios, que cuando sumamos *uno* más *uno* el resultado es *cero* y *nos llevamos uno*. Para simplificar y acercarnos a la terminología del Z80 y su juego de instrucciones, vamos a llamar **carry** al hecho de **llevarnos una**, por tanto, y siguiendo este acuerdo, *1* + *1* es igual 0 y *carry*. Y en el caso de **no llevarnos nada**, diremos **no carry**. Un ejemplo en este sentido podría ser *1* + 0 es igual a *1* y *no carry*.

¿Entendido?

Bien, pues aclarado esto sigamos con *carry* y sumemos dos números binarios de 8 dígitos cuyo resultado sea inferior a 255:

0 1 0 1 0 0 1 1 B	equivalente a	83 decimal
+ 0 0 1 0 1 1 0 1 B	equivalente a	45 decimal
1 0 0 0 0 0 0 0	equivalente a	128 decimal

En todas las columnas de esta suma, excepto en la última, hemos tenido *carry*, con lo cual los 8 bits del byte nos han sido suficientes.

Veamos un ejemplo de suma cuyo resultado sea superior a 255:

0 1 0 1 1 1 1 1 B	→	equivalente a	95 decimal
+ 1 1 1 0 0 1 1 0 B	→	equivalente a	230 decimal
0 1 0 0 0 1 0 1 B	→	no equivalente a	325 decimal
carry ↙ 69	↘	sí equivalente a	1 × 256 + 69 = 325

En este caso se ha sobrepasado el campo numérico de un byte –que como sabemos está entre 0 0 0 0 0 0 0 0 y 1 1 1 1 1 1 1 1– y necesitaríamos un bit más para acoger el procedente del *carry*. Por esta razón, el registro de señalizaciones nos provee de la *bandera* **c**, la cual tendrá el valor 1 si hay *carry* como en el caso anterior.

La *bandera* **c**, es denominada también *carry flag*. Esta situación de *carry* o *no carry*, se puede dar igualmente al restar dos números.

Veamos un ejemplo de *no carry*:

1 1 1 0 0 1 1 0 B	equivalente a	230 decimal
– 1 1 1 0 0 0 0 1 B	equivalente a	225 decimal
0 0 0 0 0 1 0 1	equivalente a	5 decimal

Pero si el minuendo fuera inferior al sustraendo

	1 1 1 0 0 0 0 1 B	equivalente a	225 decimal
−	1 1 1 0 0 1 1 0 B	equivalente a	230 decimal
	1 1 1 1 1 0 1 1 →	no equivalente a	−5 decimal
carry ↙	250	↘ Si equivalente a	251 − 1 X 256 = −5

Como consecuencia de lo estudiado sobre *carry* y *no carry*, aparece como evidente el par de nemónicos.

jp c, *NN* **jp nc**, *NN*

jp c, *NN*: Salta a la dirección de memoria NN si *carry*.

jp nc, *NN*: Salta a la dirección de memoria NN si *no carry*.

Por supuesto la dirección *NN* puede estar representada por una etiqueta.

¿Se ha fijado que siempre hemos trabajado con números comprendidos entre 0 y 255?

¿Es que un microprocesador no puede manejar cantidades superiores?

Un poco más de matemáticas. 16 bits

En el capítulo **PUESTA EN MARCHA** vimos cómo el microprocesador utiliza dos direcciones de memoria para ampliar su campo numérico de 0-255 a 0-65535. La primera dirección de memoria *–la más baja–* cuenta de uno en uno y hasta 255 (LSB) la segunda dirección de memoria *–la más alta–* cuenta de 256 en 256 (MSB), puesto que cada vez que aumenta en 1, representa que la anterior dirección ha pasado de 0 a 255.

Observe, por otra parte, que *alto* en inglés se escribe **h***ight* y *bajo* equivale a **l***ow*.

Dicho todo lo anterior, es fácil deducir que el par de registros **hl** al recibir un número superior a 255, se rellenará con el *MSB* en el registro **h** y con el *LSB* en el **l**.

Un criterio similar siguen los pares de registros **bc** y **de**.

Por todo ello, cuando deseemos copiar el contenido de un par de registros a otro, tendríamos que transferir la información de un registro a su equivalente. Supongamos, por ejemplo, que queremos cargar el par de registros **hl** con el contenido de **bc**, el proceso sería:

```
ld h,b
ld l,c
```

con lo cual las posiciones MSB y el LSB han sido transferidas en el orden conveniente.

Con el nemónico **ex bc,hl** intercambiamos el contenido de ambos pares de registros.

Con **ld rr,***NN* cargamos cualquier par de registros (**rr**) con el número *NN*, el cual puede estar comprendido entre 0 y *65535*.

Con **ld rr,(***NN***)** cargamos cualquier par de registros (**rr**) con el contenido de dos posiciones de memoria cuya primera dirección es *NN*.

Con **ld (NN),** *rr* cargamos dos posiciones de memoria cuya primera dirección es *NN* con el contenido del par de registros **rr**.

El par de registros **hl** y su posibilidad de usarlo como una especie de "puntero" ha sido usada reiteradamente en la mayoría de los ejercicios y es conocida por el lector.

Visto esto, pasemos a la aritmética de los números conformados por dos bytes –16 bits– y, por tanto, comprendidos entre 0 y 65535.

Los nemónicos que intervienen, al nivel didáctico admitido hasta el momento, son los siguientes:

add hl,bc
add hl,dc
add hl,hl
add hl,sp

Con estas instrucciones sumamos al contenido del par de registros **hl** el contenido del par de registros que se indica en cada caso.

adc hl,bc
adc hl,hl
adc hl,hl
adc hl,sp

Estas instrucciones son idénticas a las anteriores pero añadiendo el *carry* al registro **hl**, lo cual permitirá operaciones de 32 bits.

Las operaciones con 32 bits –y bytes– exigirían la siguiente operación para determinar el número decimal correspondiente:

*Contenido byte más bajo + 256 * Contenido byte inmediatamente superior + 256 * 256 * Contenido byte inmediatamente superior + 256 * 256 * 256 * Contenido del byte más alto.*

inc bc
inc de
inc hl
inc sp

Con estas instrucciones *incrementamos en uno* el contenido de los pares de registros que aparecen en cada caso.

dec bc
dec de
dec hl
dec sp

Con estas instrucciones se *decrementan en uno* el contenido de los pares de registros que aparecen en cada caso.

sbc hl,bc
sbc hl,de
sbc hl,hl
sbc hl,sp

Estas instrucciones restan al contenido del par de registros **hl** el contenido del par de registros que aparece en cada caso pero sustrayendo el *carry* del contenido del **hl**.

and, or y **xor** deja el valor del *bit* 0 del **registro f** –*carry flag*– a 0.

scf deja el *carry flag* a 1.

cct nos cambia el valor del *carry flag*.

Basta de Matemáticas.

Ya estamos en condiciones de manipular bits.

Manipulando bits

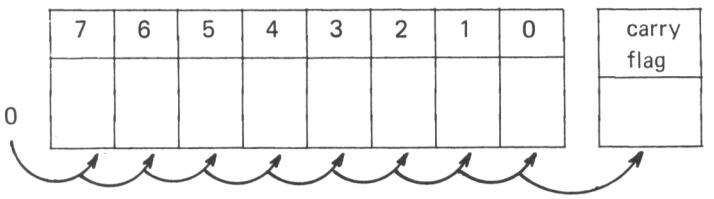
Para entrar en el tema debemos recordar el acuerdo por el cual los bits de un byte se numeran de esta forma

7	6	5	4	3	2	1	0	posición del bit
								contenido (0 ó 1)

Y a continuación estudiar los *desplazamientos lógicos y aritméticos.*

Las instrucciones de desplazamiento mueven los bits de un byte situado en un registro, o en una posición de memoria, un lugar a la derecha o un lugar a la izquierda.

Las posibilidades que se nos ofrecen son



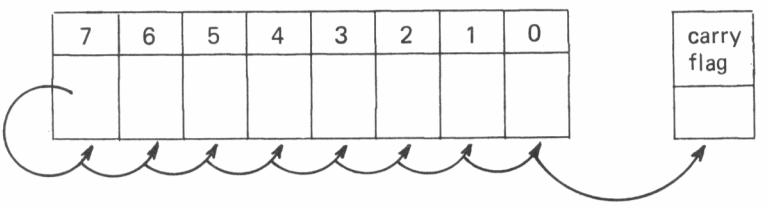
Todos los bits mueven su valor una posición a la derecha, pasando un 0 a ocupar el bit 7 y el contenido del bit 0 pasa al *carri flag*.

Este tipo de desplazamiento se denomina **srl**, que son las iniciales de *Shift Right Logical.* La sintaxis de esta instrucción es

srl r

Siendo **r** cualquier registro de 8 bits o el contenido de una posición de memoria cuya dirección esté controlada gracias al par **h l**.

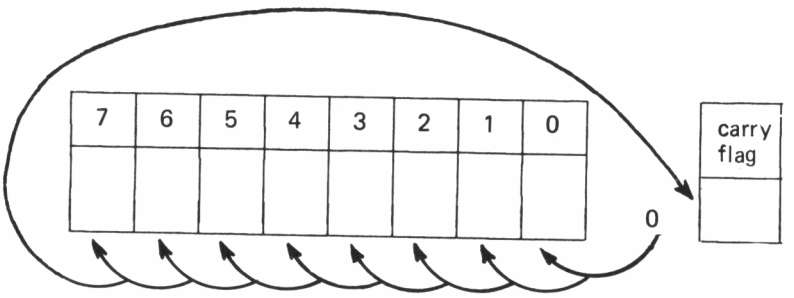
Otro desplazamiento es *Shift Right Arithmetic* - **sra** y su modelo es



Todos los bits mueven su valor una posición a la derecha, pasando el contenido del bit ϕ al *carry flag* y el contenido del bit 7 vuelve a rellenar el mismo bit. Su sintaxis es **sra r**

r guarda el mismo significado

El último desplazamiento responde al siguiente esquema:



Todos los bits mueven su contenido una posición a la izquierda, pasando un 0 al bit 0 y el contenido del bit 7 al *carry flag*.

Su denominación es *Shift Left Arithmetic* –**sla**– y la sintaxis:

sla r

r guarda el mismo significado.

Observemos estos ejemplos.

1.° Hagamos un **sra** sobre el número 00010101 y obtendremos.

	carry
0 0 0 0 1 0 1 0	1

Si este número lo multiplicamos por dos, o lo que es igual, lo sumamos asimismo tendremos

	carry
0 0 0 0 1 0 1 0	
0 0 0 0 1 0 1 0	1
0 0 0 1 0 1 0 0	
0 0 0 0 0 0 0 1	
0 0 0 1 0 1 0 1	→ número sobre el que aplicamos el **sra**

Como podemos ver en este ejemplo, un **sra** *sobre un byte equivale a dividirlo por 2, quedando el resto en el carry flag.*

2.° Hagamos un **sla** sobre el número 0 0 0 1 0 1 0 1 y se obtendrá.

	carry
0 0 1 0 1 0 1 0	0

y este número es, a su vez, el resultado de multiplicar por 2 el número inicial.

$$\begin{array}{r} \phi\,\phi\,\phi\,1\,\phi\,1\,\phi\,1 \\ +\ \phi\,\phi\,\phi\,1\,\phi\,1\,\phi\,1 \\ \hline \phi\,\phi\,1\,\phi\,1\,\phi\,1\,\phi \end{array}$$

→ número sobre el que aplicamos el **sla**

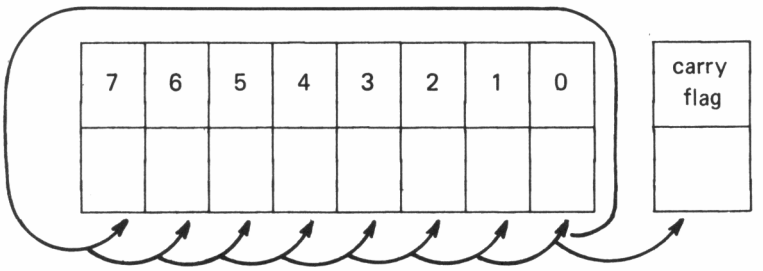
o, dicho de otro modo, *una* **sla** *sobre un byte equivale a multiplicarlo por 2.*

Las *rotaciones* son prácticamente iguales a los *desplazamientos,* con la única diferencia que el bit que sale del byte por un textremo entra otra vez en el mismo byte por el otro extremo, con las siguientes particularidades.

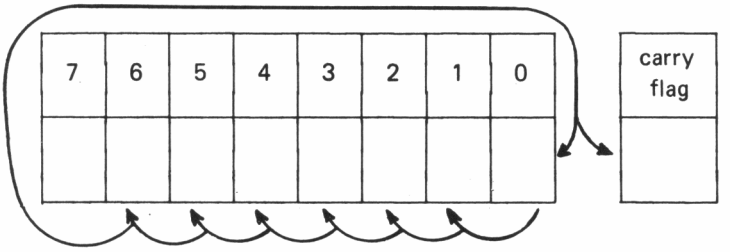
Rotaciones circulares

En estas rotaciones el bit que *sale*, pasa al otro extremo del byte y, además, se copia en el *carry flag*. Existen dos tipos:

– rotación circular a la derecha - **rrca**



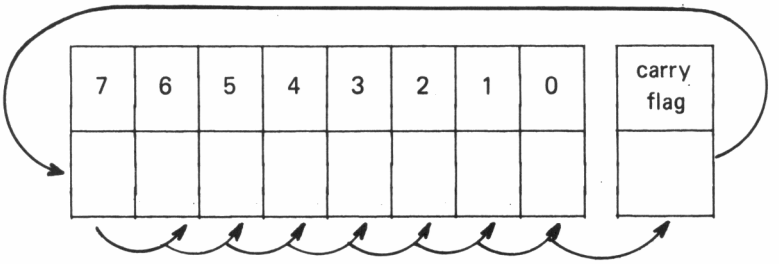
– rotación circular a la izquierda - **rlca** -



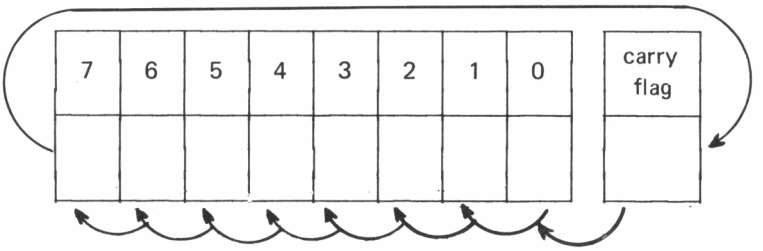
Rotaciones circulares a través del carry

En estas rotaciones el bit que *sale*, pasa al *carry flag* y el contenido del *carry flag* pasa al extremo opuesto. Existen dos tipos:

– a la derecha - **rra** -



– a la izquierda - **rla** -



Todas estas instrucciones están referidas al acumulador y, consiguientemente, son rotaciones dentro del registro **a**.

Para el resto de los registros de 8 bits se necesita un operando que especifique en cuál de ellos –o en qué dirección de memoria– se va a producir la rotación.

En estos casos la sintaxis de las instrucciones responde a:

– Rotación circular a la derecha: **rrc r**.

– Rotación circular a la izquierda: **rlc r**.

– Rotación a la derecha a través del *carry:* **rr r**.

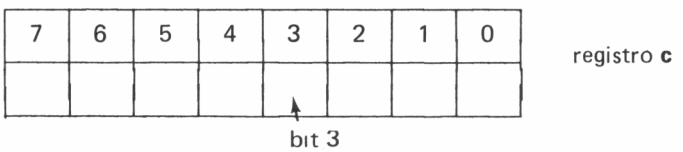
– Rotación a la izquierda a través del *carry:* **rl r**.

Para finalizar este capítulo, veremos cómo actuar sobre un bit determinado dentro de un byte. El byte debe estar en un registro o en una posición de memoria.

Cuando nos encontremos con una instrucción de este tipo:

bit n , r

la debemos interpretar como una referencia al bit situado en la posición **n** dentro del byte situado en el registro –o dirección de memoria– definido por **r**. Así, **bit 3,c** nos indica



Y significa *contenido del bit 3 en el registro c* y según sea el valor de ese bit –ϕ o 1– podremos imponer condiciones de salto.

También podemos fijar el valor de un determinado bit, dentro de un byte, gracias a las instrucciones

set n, r

res n, r

en ambos casos **n** y **r** mantienen el significado dado anteriormente y, en conjunto, la interpretación de las instrucciones es

set n,r - deja a 1 el contenido del bit **n** dentro del byte situado en **r**.

res n,r - deja a 0 el contenido del bit **n** dentro del byte situado en **r**.

Los bytes contenidos en cualquier dirección de memoria, deben ser localizados, a través del par **hl**.

Entre las páginas 186 y 187 del Manual del Spectrum, aparece un grupo de nemónicos que de alguna forma nos son familiares, bien por el BASIC, bien por el Sr. Boole y su álgebra.

and xor or

Con estas instrucciones producimos *combinaciones lógicas* entre los bits contenidos en el acumulador y otros 8 bits. Estos otros 8 bits pueden ser el contenido de una posición de memoria dada por **(hl)** o un número binario de 8 dígitos.

El resultado de la combinación lógica queda en **a**.

Estas combinaciones pueden ser:

– Impuestas por **and** (*Y* lógico)

En primer lugar debemos recordar su tabla de verdad aplicada a los valores implicados.

X (un bit situado en el acumulador)	Y (un bit del mismo peso del byte a combinar)	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

Ejemplo:

Contenido del acumulador 0 1 0 1 0 1 0 1
bits a combinar 1 1 1 1 0 1 0 1

Contenido acumulador después de **and** . . 0 1 0 1 0 1 0 1

– Impuestas por **xor** (o lógico exclusivo)

Tabla de la verdad aplicada a los valores implicados

X (un bit situado en el acumulador)	Y (un bit del mismo peso del byte a combinar)	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

Ejemplo:

Contenido del acumulador 0 1 0 1 0 1 0 1
bits a combinar 1 1 1 1 0 1 0 1

Contenido acumulador después de **xor** . . 1 0 1 0 0 0 0 0

– Impuestas por **or** (ó lógico)

Tabla de la verdad aplicada a los valores implicados

X (un bit situado en el acumulador)	Y (un bit del mismo peso del byte a combinar)	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

Ejemplo:

Contenido del acumulador 0 1 0 1 0 1 0 1
bits a combinar 1 1 1 1 0 1 0 1

Contenido acumulador después de **or** ... 1 1 1 1 0 1 0 1

– Impuestas por **not** (no lógico). Nemónico **cpl**

X (un bit situado en el acumulador)	not X
0	1
1	0

Ejemplo:

Contenido del acumulador 0 1 0 1 0 1 0 1
Contenido acumulador después de **not** .. 1 0 1 0 1 0 1 0

La sintaxis de estas instrucciones se deduce de lo dicho hasta aquí, puesto que el acumulador es el registro donde se produce la combinación lógica y, además, donde queda el resultado.

Esto quiere decir que basta con un operando a continuación de los nemónicos **and**, **xor** y **or** y ningún operando para **cpl**.

Las conclusiones y posibilidades que se derivan de los últimos capítulos son enormes especialmente a nivel matemático y sólo Vd. mismo debe decidir si puede y quiere profundizar en el tema.

Penúltimo

Reiteradamente se ha hecho referencia a las páginas 183, 184, 185, 186, 187 y 188 del Manual del Spectrum y a la columna donde figuran las instrucciones del lenguaje de asamblea del microprocesador **Z80**. Tal vez se hayan punteado aquellos nemónicos que fueron comentados a lo largo de estas páginas. Si es así, podrá comprobar que prácticamente todos fueron estudiados.

Este capítulo PENULTIMO está dedicado a tomar contacto con las restantes.

daa

Esta instrucción está dedicada a tratar con números binarios codificados en BCD (Binary Coded Decimal) y sólo opera sobre el acumulador.

Es de aplicación para pasar de binario a decimal y posterior salida a pantalla.

halt no es muy usual. Detiene al proceso.

out

Esta instrucción la utilizaremos cada vez que necesitemos enviar información al mundo exterior al Spectrum y que, de alguna forma, esté conectado con él, por ejemplo: altavoz, etc.

Todos estos periféricos se relacionan con el computador a través de puertas específicas.

La sintaxis de esta instrucción responde a

out *(puerto)*,**a**

puerto representa el número de puerto por la cual se va a enviar el byte contenido en **a**.

in a,(*puerto*)

Con esta instrucción transferimos el byte procedente del puerto, número de *puerto*, al acumulador.

exx

Con esta instrucción pasamos a usar los registros alternativos, dejando los principales y, a partir de **exx**, todos los procesos se refieren a estos registros.

Para volver a los registros principales, aplicaríamos nuevamente esta instrucción.

rst

Anteriormente se ha usado **rst 16** como una llamada a una subrutina situada en la ROM para cubrir una función determinada.

Un Z80 tiene 8 instrucciones de llamadas a subrutinas situadas en las siguientes direcciones y que permiten el uso de la instrucción **rst**:

0, 8, 16, 24, 32, 40, 48 y 56

Siendo absolutamente equivalente, por ejemplo, un **rst 16** a un **call 16**, pero con la diferencia de que en el primer caso consumimos una posición de memoria y en el segundo tres.

Todo comienza aquí

En la primera página del libro me refería a su voluntad de aprender y a mi deseo de mostrarle el camino.

Hacia la mitad del curso le avisaba de que el control de su propio aprendizaje comenzaba a depender de su iniciativa y su imaginación.

Aquí, al finalizar, es evidente que ya no podré hacer nada que no esté hecho en el camino para aumentar sus conocimientos sobre esta forma de programar.

No obstante, me resisto a dejar estas horas de secreta camaradería que el código máquina nos ha deparado, sin darle unos consejos de última hora y proporcionarle caminos para investigar conceptos.

Va de consejos

1.º Lea todo lo que caiga en sus manos referente a estos temas y los relacionados con las aplicaciones que le interesen.

2.º Considere el código máquina como un poderoso auxiliar –en forma de subrutina– de sus programas en BASIC.

3.º No se menosprecie y escriba. Escrita artículos sobre pequeñas y grandes cosas. Mande programas a las revistas. Subrutinas que Vd. piense que son prácticas...

Alguien se lo agradecerá.

...y, ¡recuerde! aunque "otros" hayan hecho mucho, Vd. lo puede hacer mejor.

Va de investigación

Cada vez que necesite conseguir algo del microprocesador tiene que saber en primer lugar si lo puede hacer y en segundo lugar qué técnica se debe seguir para lograrlo.

Supongamos que deseamos que el Spectrum emita un sonido cuando la puerta de su habitación se abra. El Z80 está en condiciones de hacer sonar el altavoz que el Spectrum lleva incorporado, pero no está conectado en forma alguna con las puertas, las bombillas o la radio.

Para tener la posibilidad de lograr el hipotético proceso propuesto, tendríamos que colocar algún dispositivo en la puerta, de forma tal, que al abrirse, enviara una señal al microprocesador, el cual, al percibir ese señal, actuaría según un programa que, finalmente, haría sonar "España Cañí" pongamos por caso.

Desarrollar este ejemplo requiere otros conocimientos y medios. Nosotros, pues, vamos a limitarnos a llegar hasta donde podemos, logrando un pequeño ruidito procedente del ordenador –que tal vez le abra la puerta a mayores empresas.

Solucionar éste y otros muchos y diferentes problemas no presenta dificultad si conocemos algún camino para acceder a las subrutinas situadas en la ROM. Un resumen de las más interesantes se dan en una tabla al final del libro y a ella nos tendremos que dirigir para escribir el listado del programa que active el altavoz.

En la columna **equ** de la *Tabla de Subrutinas en la ROM*, encontrará la expresión **Nota** que es el nombre que yo he asignado a la subrutina que se inicia en la dirección 949, si Vd. prefiere, puede darle el nombre que le resulte más evocador de la función que sumple y que está explicada en la segunda columna de la citada tabla.

La forma de hacer operativa esta subrutina está esbozada en la columna **OBSERVACIONES.**

El siguiente ejemplo simula una sirena que funcionará ininterrumpidamente hasta que se aprieta la tecla de BREAK.

Ejercicio "32"

```
    1 REM 00000000000000000000000
00000000000000000000000000000000
   10 REM go
   20 REM org 23760
   30 REM Test;call 703;cp 32;jp
z,Stop
   50 REM ld de,250
   60 REM ld hl,1000
   70 REM call 949
   80 REM ld de,100;ld hl,1500
   90 REM call 949
  100 REM jp Test
  110 REM Stop
  500 REM ret
  510 REM finish
  520 RANDOMIZE USR 60000
  530 RANDOMIZE USR 23760


org 23760
Test
23760 CD BF 02      call 703
23763 FE 20         cp 32
23765 CA ED 5C      jp z,Stop
23768 11 FA 00      ld de,250
23771 21 E8 03      ld hl,1000
23774 CD B5 03      call 949
23777 11 64 00      ld de,100
23780 21 DC 05      ld hl,1500
23783 CD B5 03      call 949
23786 C3 D0 5C      jp Test
Stop
23789 C9            ret
```

A continuación encontrará tablas que le permitirán, en combinación con todo lo estudiado, averiguar nuevas formas para solucionar asuntos ya tratados y, muy probablemente, entreverá otros campos fascinantes hacia donde dirigir sus inquietudes.

Conclusión

En la introducción a mi libro *Cómo programar su Spectrum* decía que "cuando la imaginación latina despierte al mundo de la programación, tendremos la oportunidad de recrearnos con maravillosos juegos, apoyarnos en excelentes utilidades y desarrollar nuestro trabajo diario manejando programas escritos en nuestra lengua".

Hoy podemos ver un juego como La Pulga, desarrollado por españoles, en los primeros puestos de las listas de ventas inglesas, muy buenos programas de aplicaciones tales como contabilidades, cálculos técnicos, etc., confeccionados para ordenadores populares y todo ello, aunque pueda y deba ser un motivo de orgullo, significa que una nueva actividad económica está sentando plaza y –¡qué caramba!– con la ayuda de todos nosotros, tal vez, la programación se convierta en una fuente de ingresos, vendiendo imaginación a cambio de divisas.

Bien, en todo caso, deseo que lo expuesto le haya sido útil. Si el contenido del libro, al llegar a este punto, le *ha sabido a poco* entonces me doy por satisfecho.

Hasta pronto.

Apéndices

SUBRUTINAS EN LA ROM

Denomine las subrutinas como crea conveniente

Dirección origen	Función	equ	Comentarios
703	Otea el teclado para saber si alguna tecla es apretada.	*Test*	Llamar a *Test.* El códico ASCII se carga en el acumulador.
949	Activa el altavoz emitiendo una nota determinada, durante un tiempo fijado.	*Nota*	Cargar **hl** con la nota. Cargar **de** con la duración. Llamar a *nota.*
3435	Limpia la pantalla.	*Limpia*	Abrir canal **ld a, 2; call 5633.** Llamar a *Limpia.*
3584	Desliza el contenido de la pantalla, tantas líneas hacia arriba como se determine.	*Scroll*	Cargar **b** con el número de líneas a deslizar menos uno. Llamar a *Scroll.*
3652	Borra L líneas contando desde abajo.	*Borra L*	Cargar **b** con L. Llamar a *Borra L.*
5633	Abre canal para permitir la impresión de caracteres en la pantalla.	*Abrir*	Cargar **a** con 2. Llamar a *Abrir.*
ret 16	Imprimir en pantalla el carácter cuyo código ASCII esté contenido en el acumulador. También maneja los códigos de control. Ver tabla.		Cargar en **a** el código correspondiente. Ejecutar con *rst 16.*
8859	Fija el color de BORDER.	*Borde*	Cargar **a** con el número del color elegido. Llamar a *Borde.*
8933	Fija un *pixel* en la posición x,y.	*Punto*	Cargar en **b** el valor de y. Cargar en **c** el valor de x. Llamar a *Punto.*

ATRIBUTOS

Los atributos están definidos en la dirección de memoria 23693. Un PRINT PEEK 23693 nos da 56 decimal al inicializar el sistema, lo cual equivale a

7	6	5	4	3	2	1	ϕ
0	0	1	1	1	0	0	0

Con el bit 7, se controla la situación de FLASH.

Con el bit 6, se controla la situación de BRIGHT.

Con los bits 5, 4 y 3, se controla el color de PAPER.

Con los bits 2, 1 y 0, se controla el color de INK.

Por tanto, al conectar el Spectrum, la situación de los atributos es:

bit 7, contenido 0 equivale a FLASH 0 (desactivado).

bit 6, contenido 0 equivale a BRIGHT 0 (desactivado).

bits 5, 4, 3, contenido 111 equivale a 7 igual a PAPER 7.

bits 2, 1, 0, contenido 000 equivale a 0 igual a INK 0.

Con los nemónicos **set n,(r)** y **res n,(r)** podemos dejar a 1 y 0, respectivamente el contenido del *bit n*, dentro del byte contenido en la dirección de memoria (r). En nuestro caso, la dirección de memoria es 23693 y nos debemos referir a ella a través del par de registros **hl**.

CODIGOS DE CONTROL

Estos códigos exigen tener abierto el canal de impresión

Códigos	Función	Comentarios
6	Mueve cursor a media pantalla	Cargar a con 6; **rst 16**
8	Mueve cursor a la izquierda	Cargar a con 8; **rst 16**
10	Mueve cursor abajo	Cargar a con 10; **rst 16**
11	Mueve cursor arriba	Cargar a con 11; **rst 16**
13	Mueve cursor a siguiente línea	Cargar a con 13; **rst 16**
16	Fija color de INK	Cargar a con 16; **rst 16** Cargar a con color elegido; **rst 16**
17	Fija color de PAPER	Cargar a con 17; **rst 16** Cargar a con color elegido; **rst 16**
18	Fija situación de FLASH	Cargar a con 18; **rst 16** Cargar a con 0 ó 1; **rst 16**
19	Fija situación de BRIGHT	Cargar a con 19; **rst 16** Cargar a con 0 ó 1; **rst 16**
20	Fija situación INVERSE	Cargar a con 20; **rst 16** Cargar a con 0 ó 1; **rst 16**
21	Hace actuar OVER	Cargar a con 21; **rst 16** Cargar a con 0 ó 1; **rst 16**
22	Posiciona cursor con AT	Cargar a con 22; **rst 16** Cargar a con número línea; **rst 16** Cargar a con número columna; **rst 16**
23	Posiciona cursor con TAB	Cargar a con 23; **rst 16** Cargar a con número columna; **rst 16**
32	Cursor a la derecha	Cargar a con 32; **rst 16**

JUEGO DE INSTRUCCIONES DEL Z 80

Nemónico		Nemónico		Nemónico	
Operador	Operandos	Operador	Operandos	Operador	Operandos
adc	a, (hl)	and	h	bit	4, a
adc	a, (ix + ind)	and	l	bit	4, b
adc	a, (iy + ind)	and	n	bit	4, c
adc	a, a	bit	o, (hl)	bit	4, d
adc	a, b	bit	o, (ix + ind)	bit	4, e
adc	a, c	bit	o, (iy + ind)	bit	4, h
adc	a, d	bit	o, a	bit	4, l
adc	a, e	bit	o, b	bit	5, (hl)
adc	a, h	bit	o, c	bit	5, (ix + ind)
adc	a, l	bit	o, d	bit	5, (iy + ind)
adc	a, n	bit	o, e	bit	5, a
adc	hl, bc	bit	o, h	bit	5, b
adc	hl, de	bit	o, l	bit	5, c
adc	hl, hl	bit	1, (hl)	bit	5, d
adc	hl, sp	bit	1, (ix + ind)	bit	5, e
add	a, (hl)	bit	1, (iy + ind)	bit	5, h
add	a, (ix + ind)	bit	1, a	bit	5, l
add	a, (iy + ind)	bit	1, b	bit	6, (hl)
add	a, a	bit	1, c	bit	6, (ix + ind)
add	a, b	bit	i, d	bit	6, (iy + ind)
add	a, c	bit	1, e	bit	6, a
add	a, d	bit	1, h	bit	6, b
add	a, e	bit	1, l	bit	6, c
add	a, h	bit	2, (hl)	bit	6, d
add	a, l	bit	2 (ix + ind)	bit	6, e
add	a, n	bit	2, (iy + ind)	bit	6, h
add	hl, bc	bit	2, a	bit	6, l
add	hl, de	bit	2, b	bit	7, (hl)
add	hl, hl	bit	2, c	bit	7, (ix + ind)
add	hl, sp	bit	2, d	bit	7, (iy + ind)
add	ix, bc	bit	2, e	bit	7, a
add	ix, de	bit	2, h	bit	7, b
add	ix, ix	bit	2, l	bit	7, c
add	ix, sp	bit	3, (hl)	bit	7, d
add	iy, bc	bit	3, (ix + ind)	bit	7, e
add	iy, de	bit	3, (iy + ind)	bit	7, h
add	iy, iy	bit	3, a	bit	7, l
add	iy, sp	bit	3, b	call	c, nn
and	(hl)	bit	3, c	call	m, nn
and	(ix + ind)	bit	3, d	call	nc, nn
and	(iy + ind)	bit	3, e	call	nn
and	a	bit	3, h	call	nz, nn
and	b	bit	3, l	call	p, nn
and	c	bit	4, (hl)	call	pe, nn
and	d	bit	4, (ix + ind)	call	po, nn
and	e	bit	4, (iy + ind)	call	z, nn

Nemónico		Nemónico		Nemónico	
Operador	Operandos	Operador	Operandos	Operador	Operandos
ccf		in	e, (c)	ld	(ix + ind), c
cp	(hl)	in	h, (c)	ld	(ix + ind), d
cp	(ix + ind)	in	l, (c)	ld	(ix + ind), e
cp	(iy + ind)	inc	(hl)	ld	(ix + ind), h
cp	a	inc	(ix + ind)	ld	(ix + ind), l
cp	b	inc	(iy + ind)	ld	(ix + ind), n
cp	c	inc	a	ld	(iy + ind), a
cp	d	inc	b	ld	(iy + ind), b
cp	e	inc	bc	ld	(iy + ind), c
cp	h	inc	c	ld	(iy + ind), d
cp	l	inc	d	ld	(iy + ind), e
cp	n	inc	de	ld	(iy + ind), h
cpd		inc	e	ld	(iy + ind), l
cpdr		inc	h	ld	(iy + ind), n
cpi		inc	hl	ld	(nn), a
cpir		inc	ix	ld	(nn), bc
cpl		inc	iy	ld	(nn), de
daa		inc	l	ld	(nn), hl
dec	(hl)	inc	sp	ld	(nn), ix
dec	(ix + ind)	ind		ld	(nn), iy
dec	(iy + ind)	indr		ld	(nn), sp
dec	a	ini		ld	a, (bc)
dec	b	inir		ld	a, (de)
dec	bc	jp	(hl)	ld	a, (hl)
dec	c	jp	(ix)	ld	a, (ix + ind)
dec	d	jp	(iy)	ld	a, (iy + ind)
dec	de	jp	c, nn	ld	a, (nn)
dec	e	jp	m, nn	ld	a, a
dec	h	jp	nc, nn	ld	a, b
dec	hl	jp	nn	ld	a, c
dec	ix	jp	nz, nn	ld	a, d
dec	iy	jp	p, nn	ld	a, e
dec	l	jp	pe, nn	ld	a, h
dec	sp	jp	po, nn	ld	a, i
di		jp	z, nn	ld	a, l
djnz	dis	jr	c, dis	ld	a, n
ei		jr	dis	ld	b, (hl)
ex	(sp), hl	jr	nc, dis	ld	b, (ix + ind)
ex	(sp), ix	jr	nz, dis	ld	b, (iy + ind)
ex	(sp), iy	jr	z, dis	ld	b, a
ex	af, af	ld	(bc), a	ld	b, b
ex	de, hl	ld	(de), a	ld	b, c
exx		ld	(hl), a	ld	b, d
halt		ld	(hl), b	ld	b, e
im	0	ld	(hl), c	ld	b, h, nn
im	1	ld	(hl), d	ld	b, l
im	2	ld	(hl), e	ld	b, n
in	a, (c)	ld	(hl), h	ld	bc, (nn)
in	a, n	ld	(hl), l	ld	bc, nn
in	b, (c)	ld	(hl), n	ld	c, (hl)
in	c, (c)	ld	(ix + ind), a	ld	c, (ix + ind)
in	d, (c)	ld	(ix + ind), b	ld	c, (iy + ind)

Nemónico		Nemónico		Nemónico	
Operador	Operandos	Operador	Operandos	Operador	Operandos
ld	c, a	ld	l,(iy + ind)	push	hl
ld	c, b	ld	l,a	push	ix
ld	c, c	ld	l,b	push	iy
ld	c, d	ld	l,c	res	0,(hl)
ld	c, e	ld	l,d	res	0,(ix + ind)
ld	c, h	ld	l,e	res	0,(iy + ind)
ld	c, l	ld	l,h	res	0,a
ld	c, n	ld	l,l	res	0,b
ld	d, (hl)	ld	l,n	res	0,c
ld	d, (ix + ind)	ld	sp,(nn)	res	0,d
ld	d, (iy + ind)	ld	sp,hl	res	0,e
ld	d, a	ld	sp,ix	res	0,h
ld	d, b	ld	sp,iy	res	0,l
ld	d, c	ld	sp,nn	res	1,(hl)
ld	d, d	ldd		res	1,(ix + ind)
ld	d, e	lddr		res	1,(iy + ind)
ld	d, h	ldi		res	1,a
ld	d, l	ldir		res	1,b
ld	d, n	neg		res	1,c
ld	de, (nn)	neop		res	1,d
ld	de, nn	or	(hl)	res	1,e
ld	e, (hl)	or	(ix + ind)	res	1,h
ld	e, (ix + ind)	or	(iy + ind)	res	1,l
ld	e, (iy + ind)	or	a	res	2,(hl)
ld	e, a	or	b	res	2,(ix + ind)
ld	e, b	or	c	res	2,(iy + ind)
ld	e, c	or	d	res	2,a
ld	e, d	or	e	res	2,b
ld	e, e	or	h	res	2,c
ld	e, h	or	l	res	2,d
ld	e, l	or	n	res	2,e
ld	e, n	otdr		res	2,h
ld	h, (hl)	otir		res	2,l
ld	h, (ix + ind)	out	(c),a	res	3,(hl)
ld	h, (iy + ind)	out	(c),b	res	3,(ix + ind)
ld	h, a	out	(c),c	res	3,(iy + ind)
ld	h, b	out	(c),d	res	3,a
ld	h, c	out	(c),e	res	3,b
ld	h, d	out	(c),h	res	3,c
ld	h, e	out	(c),l	res	3,d
ld	h, h	out	n,a	res	3,e
ld	h, l	outd		res	3,h
ld	h, n	outi		res	3,l
ld	hl, (nn)	pop	af	res	4,(hl)
ld	hl, nn	pop	bc	res	4,(ix + ind)
ld	i, a	pop	de	res	4,(iy + ind)
ld	ix, (nn)	pop	hl	res	4,a
ld	ix, nn	pop	ix	res	4,b
ld	iy, (nn)	pop	iy	res	4,c
ld	iy, nn	push	af	res	4,d
ld	l, (hl)	push	bc	res	4,e
ld	l,(ix + ind)	push	de	res	4,h

Nemónico		Nemónico		Nemónico	
Operador	Operandos	Operador	Operandos	Operador	Operandos
res	4,l	rla		sbc	a,h
res	5,(hl)	rlc	(hl)	sbc	a,l
res	5,(ix + ind)	rlc	(ix + ind)	sbc	a,n
res	5,(iy + ind)	rlc	(iy + ind)	sbc	hl,bc
res	5,a	rlc	a	sbc	hl,de
res	5,b	rlc	b	sbc	hl,hl
res	5,c	rlc	c	sbc	hl,sp
res	5,d	rlc	d	scf	
res	5,e	rlc	e	set	0,(hl)
res	5,h	rlc	h	set	0,(ix + ind)
res	5,l	rlc	l	set	0,(iy + ind)
res	6,(hl)	rlca		set	0,a
res	6,(ix + ind)	rld		set	0,b
res	6,(iy + ind)	rr	(hl)	set	0,c
res	6,a	rr	(ix + ind)	set	0,d
res	6,b	rr	(iy + ind)	set	0,e
res	6,c	rr	a	set	0,h
res	6,d	rr	b	set	0,l
res	6,e	rr	c	set	1,(hl)
res	6,h	rr	d	set	1,(ix + ind)
res	6,l	rr	e	set	1,(iy + ind)
res	7,(hl)	rr	h	set	1,a
res	7,(ix + ind)	rr	l	set	1,b
res	7,(iy + ind)	rra		set	1,c
res	7,a	rrc	(hl)	set	1,d
res	7,b	rrc	(ix + ind)	set	1,e
res	7,c	rrc	(iy + ind)	set	1,h
res	7,d	rrc	a	set	1,l
res	7,e	rrc	b	set	2,(hl)
res	7,h	rrc	c	set	2,(ix + ind)
res	7,l	rrc	d	set	2,(iy + ind)
ret		rrc	e	set	2,a
ret	c	rrc	h	set	2,b
ret	m	rrc	l	set	2,c
ret	nc	rrca		set	2,d
ret	nz	rrd		set	2,e
ret	p	rst	0	set	2,h
ret	pe	rst	10h	set	2,l
ret	po	rst	18h	set	3,b
ret	z	rst	20h	set	3,(hl)
reti		rst	28h	set	3,(ix + ind)
retn		rst	30h	set	3,(iy + ind)
rl	(hl)	rst	38h	set	3,a
rl	(ix + ind)	rst	8	set	3,c
rl	(iy + ind)	sbc	a,(hl)	set	3,d
rl	a	sbc	a,(ix + ind)	set	3,e
rl	b	sbc	a,(iy + ind)	set	3,h
rl	c	sbc	a,a	set	3,l
rl	d	sbc	a,b	set	4,(hl)
rl	e	sbc	a,c	set	4,(ix + ind)
rl	h	sbc	a,d	set	4,(iy + ind)
rl	l	sbc	a,e	set	4,a

Nemónico		Nemónico		Nemónico	
Operador	Operandos	Operador	Operandos	Operador	Operandos
set	4,b	set	7,d	srl	e
set	4,c	set	7,e	srl	h
set	4,d	set	7,h	srl	l
set	4,e	set	7,l	sub	(hl)
set	4,h	sla	(hl)	sub	(ix + ind)
set	4,l	sla	(ix + ind)	sub	(iy + ind)
set	5,(hl)	sla	(iy + ind)	sub	a
set	5,(ix + ind)	sla	a	sub	b
set	5,(iy + ind)	sla	b	sub	c
set	5,a	sla	c	sub	d
set	5,b	sla	d	sub	e
set	5,c	sla	e	sub	h
set	5,d	sla	h	sub	l
set	5,e	sla	l	sub	n
set	5,h	sra	(hl)	xor	(hl)
set	5,l	sra	(ix + ind)	xor	(ix + ind)
set	6,(hl)	sra	(iy + ind)	xor	(iy + ind)
set	6,(ix + ind)	sra	a	xor	a
set	6,(iy + ind)	sra	b	xor	b
set	6,a	sra	c	xor	c
set	6,b	sra	d	xor	d
set	6,c	sra	e	xor	e
set	6,d	sra	h	xor	h
set	6,e	sra	l	xor	l
set	6,h	srl	(hl)	xor	n
set	6,l	srl	(ix + ind)	defs	2
set	7,(hl)	srl	(iy + ind)	equ	5
set	7,(ix + ind)	srl	a	equ	10h
set	7,(iy + ind)	srl	b	equ	20h
set	7,a	srl	c	equ	30h
set	7,b	srl	d	end	
set	7,c				

INTERPRETACIONES ABREVIADAS DE NEMONICOS

adc – adición con *carry*
add – adición
and – y lógico
bit – test bits
call – llamada a subrutina
cp – comparar
cpd – comparar decrementado
cpi – comparar incrementado
daa – ajuste decimal
dec – decrementar
djnz – salto y decremento
ex – intercambio registros
exx – intercambio registros auxiliares
halt – interrupción absoluta
in – entrada de un byte
inc – incrementar
ind – entrada decrementando
ini – entrada incrementando
jp – salto
jr – salto relativo
ld – cargar
ldd – cargar memoria abajo
ldi – cargar memoria arriba
neg – negación
nop – no opera
or – o lógico
out – salida
outd – salida decrementando
outi – salida incrementando
pop – tirar, sacar
push – empujar
res – reset sobre un bit
ret – retorno
rl – rotación izquierda incluido *carry*
rlc – rotación izquierda copia a *carry*
rld – rotación izquierda en BCD
rr – rotación derecha incluido *carry*
rrc – rotación derecha copia a *carry*
rrd – rotación derecha en BCD
rst – llamada a subrutinas RESTART
sbc – sustracción con *carry*
set – un bit a 1
sla – decalaje aritmético izquierda
sra – decalaje aritmético derecha
srl – decalaje lógico derecha
sub – sustracción
xor – o exclusivo

TABLA DE CONVERSION DECIMAL Y HEXADECIMAL

Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	2's C.	Dec.	Hex.	2's C.
0	00	64	40	128	80	−128	192	C0	−64
1	01	65	41	129	81	−127	193	C1	−63
2	02	66	42	130	82	−126	194	C2	−62
3	03	67	43	131	83	−125	195	C3	−61
4	04	68	44	132	84	−124	196	C4	−60
5	05	69	45	133	85	−123	197	C5	−59
6	06	70	46	134	86	−122	198	C6	−58
7	07	71	47	135	87	−121	199	C7	−57
8	08	72	48	136	88	−120	200	C8	−56
9	09	73	49	137	89	−119	201	C9	−55
10	0A	74	4A	138	8A	−118	202	CA	−54
11	0B	75	4B	139	8B	−117	203	CB	−53
12	0C	76	4C	140	8C	−116	204	CC	−52
13	0D	77	4D	141	8D	−115	205	CD	−51
14	0E	78	4E	142	8E	−114	206	CE	−50
15	0F	79	4F	143	8F	−113	207	CF	−49
16	10	80	50	144	90	−112	208	D0	−48
17	11	81	51	145	91	−111	209	D1	−47
18	12	82	52	146	92	−110	210	D2	−46
19	13	83	53	147	93	−109	211	D3	−45
20	14	84	54	148	94	−108	212	D4	−44
21	15	85	55	149	95	−107	213	D5	−43
22	16	86	56	150	96	−106	214	D6	−42
23	17	87	57	151	97	−105	215	D7	−41
24	18	88	58	152	98	−104	216	D8	−40
25	19	89	59	153	99	−103	217	D9	−39
26	1A	90	5A	154	9A	−102	218	DA	−38
27	1B	91	5B	155	9B	−101	219	DB	−37
28	1C	92	5C	156	9C	−100	220	DC	−36
29	1D	93	5D	157	9D	−99	221	DD	−35
30	1E	94	5E	158	9E	−98	222	DE	−34
31	1F	95	5F	159	9F	−97	223	DF	−33
32	20	96	60	160	A0	−96	224	E0	−32
33	21	97	61	161	A1	−95	225	E1	−31
34	22	98	62	162	A2	−94	226	E2	−30
35	23	99	63	163	A3	−93	227	E3	−29
36	24	100	64	164	A4	−92	228	E4	−28
37	25	101	65	165	A5	−91	229	E5	−27
38	26	102	66	166	A6	−90	230	E6	−26
39	27	103	67	167	A7	−89	231	E7	−25
40	28	104	68	168	A8	−88	232	E8	−24
41	29	105	69	169	A9	−87	233	E9	−23
42	2A	106	6A	170	AA	−86	234	EA	−22
43	2B	107	6B	171	AB	−85	235	EB	−21
44	2C	108	6C	172	AC	−84	236	EC	−20
45	2D	109	6D	173	AD	−83	237	ED	−19
46	2E	110	6E	174	AE	−82	238	EE	−18
47	2F	111	6F	175	AF	−81	239	EF	−17
48	30	112	70	176	B0	−80	240	F0	−16
49	31	113	71	177	B1	−79	241	F1	−15
50	32	114	72	178	B2	−78	242	F2	−14

Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	2's C.	Dec.	Hex.	2's C.
51	33	115	73	179	B3	−77	243	F3	−13
52	34	116	74	180	B4	−76	244	F4	−12
53	35	117	75	181	B5	−75	245	F5	−11
54	36	118	76	182	B6	−74	246	F6	−10
55	37	119	77	183	B7	−73	247	F7	−9
56	38	120	78	184	B8	−72	248	F8	−8
57	39	121	79	185	B9	−71	249	F9	−7
58	3A	122	7A	186	BA	−70	250	FA	−6
59	3B	123	7B	187	BB	−69	251	FB	−5
60	3C	124	7C	188	BC	−68	252	FC	−4
61	3D	125	7D	189	BD	−67	253	FD	−3
62	3E	126	7E	190	BE	−66	254	FE	−2
63	3F	127	7F	191	BF	−65	255	FF	−1

SPECTRUM: INICIACION AL CODIGO MAQUINA

En términos de programación nada hay más allá del Código Máquina.

El Código Máquina implica hablar cara a cara con el cerebro del computador: El Microprocesador.

Si usted decide entrar en este mundo, debe creerme si le digo que, por la misma sencillez del microprocesador, este tipo de programación es una técnica sencilla.

Con frases como ésta y según su costumbre y estilo, el autor nos ayuda a *perderle el respeto* a la máquina.

Buscando más la comprensión de los conceptos y su utilización práctica, que extraer la máxima capacidad de los nuevos conocimientos, este libro es el curso fundamental por el que llegar a un correcto y cómodo entendimiento de otros más profundos y especializados.

EDITORIAL PARANINFO S.A.
Magallanes, 25 - 28015 Madrid

ISBN: 84-283-1421-7